Lecture 4.
The Java Collections Framework

Chapters 6.3-6.4
Outline

• Introduction to the Java Collections Framework
• Iterators
• Interfaces, Abstract Classes and Classes of the Java Collections Framework
Outline

• Introduction to the Java Collections Framework

• Iterators

• Interfaces, Abstract Classes and Classes of the Java Collections Framework
The Java Collections Framework

• We will consider the Java Collections Framework as a good example of how to apply the principles of object-oriented software engineering (see Lecture 1) to the design of classical data structures.
The Java Collections Framework

• A coupled set of classes and interfaces that implement commonly reusable collection data structures.

• Designed and developed primarily by Joshua Bloch (currently Chief Java Architect at Google).
What is a Collection?

• An object that groups multiple elements into a single unit.

• Sometimes called a container.
What is a Collection Framework?

• A unified architecture for representing and manipulating collections.

• Includes:
  – **Interfaces**: A hierarchy of ADTs.
  – **Implementations**
  – **Algorithms**: The methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.
    • These algorithms are *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface.
History

• Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy.
Benefits

• **Reduces programming effort**: By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work.

• **Increases program speed and quality**: Provides high-performance, high-quality implementations of useful data structures and algorithms.

• **Allows interoperability among unrelated APIs**: APIs can interoperate seamlessly, even though they were written independently.

• **Reduces effort to learn and to use new APIs**

• **Reduces effort to design new APIs**

• **Fosters software reuse**: New data structures that conform to the standard collection interfaces are by nature reusable.
Where is the Java Collections Framework?

• Package java.util.

• In this lecture we will survey the interfaces, abstract classes and classes **for linear data structures** provided by the Java Collections Framework.

• We will not cover all of the details (e.g., the exceptions that may be thrown).

• For additional details, please see
  – Javadoc, provided with your java distribution.
  – **Comments and code in the specific java.util.*.java files**, provided with your java distribution.
Core Collection Interfaces
Outline

• Introduction to the Java Collections Framework
• Iterators
• Interfaces, Abstract Classes and Classes of the Java Collections Framework
Traversing Collections in Java

• There are two ways to traverse collections:
  – using **Iterators**.
  – with the (enhanced) **for-each** construct
Iterators

• An **Iterator** is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.

• You get an Iterator for a collection by calling the collection’s iterator method.

• Suppose **collection** is an instance of a **Collection**. Then to print out each element on a separate line:

```java
Iterator<E> it = collection.iterator();
while (it.hasNext())
    System.out.println(it.next());
```

• Note that next() does two things:
  1. Returns the current element (initially the first element)
  2. Steps to the next element and makes it the current element.
Iterators

Iterator interface:

```java
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

- `hasNext()` returns true if the iteration has more elements
- `next()` returns the next element in the iteration.
  - throws exception if iterator has already visited all elements.
- `remove()` removes the last element that was returned by next.
  - remove may be called only once per call to next
  - otherwise throws an exception.
  - `Iterator.remove` is the only safe way to modify a collection during iteration
Implementing Iterators

• Could make a copy of the collection.
  – **Good**: could make copy private – no other objects could change it from under you.
  – **Bad**: construction is $O(n)$.

• Could use the collection itself (the typical choice).
  – **Good**: construction, hasNext and next are all $O(1)$.
  – **Bad**: if another object makes a structural change to the collection, the results are unspecified.
The Enhanced For-Each Statement

• Suppose `collection` is an instance of a `Collection`. Then

```
for (Object o : collection)
    System.out.println(o);
```

prints each element of the collection on a separate line.

• This code is just shorthand: it compiles to use `o.iterator()`.
The Generality of Iterators

• Note that iterators are general in that they apply to any collection.
  – Could represent a sequence, set or map.
  – Could be implemented using arrays or linked lists.
ListIterators

• A **ListIterator** extends Iterator to treat the collection as a list, allowing
  – access to the integer position (index) of elements
  – forward and backward traversal
  – modification and insertion of elements.

• The current position is viewed as being either
  – Before the first element
  – Between two elements
  – After the last element
ListIterators

- ListIterators support the following methods:
  - **add(e):** inserts element e at current position (before implicit cursor)
  - **hasNext()**
  - **hasPrevious()**
  - **previous():** returns element before current position and steps backward
  - **next():** returns element after current position and steps forward
  - **nextIndex()**
  - **previousIndex()**
  - **set(e):** replaces the element returned by the most recent **next()** or **previous()** call
  - **remove():** removes the element returned by the most recent **next()** or **previous()** call
Outline

• Introduction to the Java Collections Framework
• Iterators
• Interfaces, Abstract Classes and Classes of the Java Collections Framework
Levels of Abstraction

• Recall that Java supports three levels of abstraction:
  
  – **Interface**
    
    • Java expression of an ADT
    
    • Includes method declarations with arguments of specified types, but with empty bodies
  
  – **Abstract Class**
    
    • Implements only a subset of an interface.
    
    • Cannot be used to instantiate an object.
  
  – **(Concrete) Classes**
    
    • May extend one or more abstract classes
    
    • Must fully implement any interface it implements
    
    • Can be used to instantiate objects.
END OF LECTURE 5
JAN 21, 2014
The Java Collections Framework (Ordered Data Types)
The **Iterable** Interface

- Allows an **Iterator** to be associated with an object.
- The iterator allows an existing data structure to be stepped through sequentially, using the following methods:
  - `hasNext()` returns true if the iteration has more elements
  - `next()` returns the next element in the iteration.
    - throws exception if iterator has already visited all elements.
  - `remove()` removes the last element that was returned by next.
    - remove may be called only once per call to next
    - otherwise throws an exception.
  - `Iterator.remove` is the only safe way to modify a collection during iteration
The Collection Interface

- Allows data to be modeled as a collection of objects. In addition to the **Iterator** interface, provides interfaces for:
  - Creating the data structure
    - `add(e)`
    - `addAll(c)`
  - Querying the data structure
    - `size()`
    - `isEmpty()`
    - `contains(e)`
    - `containsAll(c)`
    - `toArray()`
    - `equals(e)`
  - Modifying the data structure
    - `remove(e)`
    - `removeAll(c)`
    - `retainAll(c)`
    - `clear()`
The Java Collections Framework (Ordered Data Types)
The Abstract Collection Class

• Skeletal implementation of the Collection interface.
• For unmodifiable collection, programmer still needs to implement:
  – iterator (including hasNext and next methods)
  – size
• For modifiable collection, need to also implement:
  – remove method for iterator
  – add
The **List** Interface

- Extends the Collections interface to model the data as an **ordered sequence** of elements, **indexed by a 0-based integer index (position)**.
- Provides interface for creation of a **ListIIterator**
- Also adds interfaces for:
  - Creating the data structure
    - `add(e)` – append element e to the list
    - `add(i, e)` – insert element e at position i (and shift elements at i and above one to the right).
  - Querying the data structure
    - `get(i)` – return element currently stored at position i
    - `indexOf(e)` – return index of first occurrence of specified element e
    - `lastIndexOf(e)` – return index of last occurrence of specified element e
    - `subList(i1, i2)` – return list of elements from index i1 to i2
  - Modifying the data structure
    - `set(i, e)` – replace element currently stored at index i with specified element e
    - `remove(e)` – remove the first occurrence of the specified element from the list
    - `remove(i)` – remove the element at position i
The Java Collections Framework (Ordered Data Types)
The **Abstract List** Class

- Skeletal implementation of the **List** interface.
- For **unmodifiable** list, programmer needs to implement methods:
  - get
  - size
- For **modifiable** list, need to implement
  - set
- For **variable-size** modifiable list, need to implement
  - add
  - remove
The Java Collections Framework (Ordered Data Types)
The **ArrayList** Class

- **Random access** data store implementation of the **List** interface
- Uses an **array** for storage.
- Supports automatic array-resizing
- Adds methods
  - `trimToSize()` – Trims capacity to current size
  - `ensureCapacity(n)` – Increases capacity to at least n
  - `clone()` – Create copy of list
  - `removeRange(i1, i2)` – Remove elements at positions i1 to i2
  - `RangeCheck(i)`: throws exception if i not in range
  - `writeObject(s)`: writes out list to output stream s
  - `readObject(s)`: reads in list from input stream s
The Vector Class

• Similar to ArrayList.

• But all methods of Vector are synchronized.
  – Uses an internal lock to prevent multiple threads from concurrently executing methods for the same vector object.
  – Other threads trying to execute methods of the object are suspended until the current thread completes.
  – Helps to prevent conflicts and inconsistencies in multi-threaded code.

• Vector is a so-called legacy class: no longer necessary for new applications, but still in widespread use in existing code.

• Synchronization can be achieved with ArrayLists and other classes of the Collections framework using synchronization wrappers (we will not cover this).
The Java Collections Framework (Ordered Data Types)
The **Stack** Class

- Represents a last-in, first-out (LIFO) stack of objects.
- Adds 5 methods:
  - `push()`
  - `pop()`
  - `peek()`
  - `empty()`
  - `search(e)`: return the 1-based position of where an object is on the stack.
The Java Collections Framework (Ordered Data Types)
The **Abstract Sequential List** Class

- Skeletal implementation of the **List** interface.
- Assumes a **sequential** access data store (e.g., **linked list**)
- Programmer needs to implement methods
  - `listIterator()`
  - `size()`
- For **unmodifiable** list, programmer needs to implement list iterator’s methods:
  - `hasNext()`
  - `next()`
  - `hasPrevious()`
  - `previous()`
  - `nextIndex()`
  - `previousIndex()`
- For **modifiable** list, need to also implement list iterator’s
  - `set(e)`
- For **variable-size** modifiable list, need to implement list iterator’s
  - `add(e)`
  - `remove()`
The Java Collections Framework (Ordered Data Types)
The **Queue** Interface

- Designed for holding elements prior to processing
- Typically first-in first-out (FIFO)
- Defines a head position, which is the next element to be removed.
- Provides additional insertion, extraction and inspection operations.
- Extends the **Collection** interface to provide interfaces for:
  - **offer(e)**: add e to queue if there is room (return false if not)
  - **poll()**: return and remove head of queue (return null if empty)
  - **remove()**: return and remove head of queue (throw exception if empty)
  - **peek()**: return head of queue (return null if empty)
  - **element()**: return head of queue (throw exception if empty)
The Java Collections Framework (Ordered Data Types)
The **LinkedList** Class

- Implements the **List** and **Queue** interfaces.
- Uses a **doubly-linked list** data structure.
- Extends the **List** interface with additional methods:
  - `getFirst()`
  - `getLast()`
  - `removeFirst()`
  - `removeLast()`
  - `addFirst(e)`
  - `addLast(e)`
- These make it easier to use the LinkedList class to create stacks, queues and deques (double-ended queues).
The LinkedList Class

- LinkedList objects are **not** synchronized by default.
- However, the LinkedList iterator is **fail-fast**: if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own remove or add methods, the iterator will throw a **ConcurrentModificationException**.
- This is detected at the first execution of one of the iterator’s methods after the modification.
- In this way the iterator will hopefully fail quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.
The Java Collections Framework (Ordered Data Types)
The **Abstract Queue** Class

- Skeletal implementation of the **Queue** interface.
- Provides implementations for
  - `add(e)`
  - `remove()`
  - `element()`
  - `clear()`
  - `addAll(c)`
The Priority Queue Class

• Based on priority heap
• Elements are prioritized based either on
  – natural order
  – a comparator, passed to the constructor.
• Provides an iterator

• We will study this in detail when we get to heaps!
The Java Collections Framework (Ordered Data Types)

- Interface
- Abstract Class
- Class

- Queue
- Abstract Queue
- Priority Queue
- Abstract Sequential List
- Array List
- Linked List
- Vector
- Stack

- Collection
- Iterable
- Abstract Collection
- Abstract List

CSE 2011
Prof. J. Elder

Last Updated: 2014-01-20 5:01 PM
Summary

• From this lecture you should understand:
  – The purpose and advantages of the Java Collections Framework
  – How interfaces, abstract classes and classes are used hierarchically to achieve some of the key goals of object-oriented software engineering.
  – The purpose of iterators, and how to create and use them.
  – How the Java Collections Framework can be used to develop code using general collections, lists, array lists, stacks and queues.
For More Details

• **Javadoc**, provided with your java distribution.

• **Comments and code in the specific java.util.*.java files**, provided with your java distribution.
