Collections and Iterators

Giuseppe Narzisi
V22.0102-003
Spring 2009
Definition

• Definition of Collection:
  – Group of objects, which are also called elements
  – May allow duplicates and requires no specific ordering

• Collection interface
  – Root of all collection interfaces
Benefits of Java Collection Framework

• Reduces programming effort
  – You don't need to create your own data structures and algorithms

• Increases program speed and quality
  – Collections framework provides high-performance, high-quality implementations of useful data structures and algorithms

• Reduces effort to learn and use new APIs

• Reduces effort to design new APIs

• Foster software reuse
The Java Collections Framework

• The classes in the Java Collections Framework fall into three general categories:

1. Lists. A list is an ordered collection of values that allows the client to add and remove elements. As you would expect, the ArrayList class falls into this category.

2. Sets. A set is an unordered collection of values in which a particular object can appear at most once.

3. Maps. A map implements an association between keys and values. The HashMap class is in this category.

• The next slide shows the Java class hierarchy for the first two categories, which together are called collections.
The Collection Hierarchy

The following diagram shows the portion of the Java Collections Framework that implements the Collection interface. The dotted lines specify that a class implements a particular interface.
**ArrayList vs. LinkedList**

- Two classes in the Java Collections Framework that implement the `List` interface: `ArrayList` and `LinkedList`.

- Because these classes implement the same interface, it is generally possible to substitute one for the other.

- These classes have the same effect, however, they do not have the same performance characteristics.
  - The `ArrayList` class is more efficient when selecting a particular element or searching for an element in a sorted array.
  - The `LinkedList` class is more efficient when adding or removing elements from a large list.

- Choosing which list implementation to use is therefore a matter of evaluating the performance tradeoffs.
Iteration in Collections

• One of the most useful operations for any collection is the ability to run through each of the elements in a loop. This process is called **iteration**.
  
  • The `java.util.Iterator<E>` interface provides for one-way traversal
  • The `java.util.ListIterator<E>` provides two-way traversal

• In older versions of Java, the programming pattern for using an iterator looks like this:

  ```java
  Iterator iterator = collection.elements();
  while (iterator.hasNext()) {
      type element = (type) iterator.next();
      ... statements that process this particular element ...
  }
  ```

• Java Standard Edition 5.0 allows you to simplify this code to

  ```java
  for (type element : collection) {
      ... statements that process this particular element ...
  }
  ```
Iteration Order

- **List**: the order in which iteration proceeds through the elements of the list is defined by the underlying ordering of the list. The element at index 0 comes first, followed by the other elements in order.

- **Set**: the ordering of iteration is more difficult to specify because a set is, by definition, an unordered collection. A set that implements only the `Set` interface is free to deliver up elements in any order.

- If a `Set` also implements the `SortedSet` interface (as the `TreeSet` class does), the iterator is forced to deliver elements that appear in ascending order according to the `compareTo` method for that class.
The Collections Toolbox

- The Collections class (not the same as the Collection interface) exports several static methods that operate on lists, the most important of which appear in the following table:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binarySearch(list, key)</code></td>
<td>Finds key in a sorted list using binary search.</td>
</tr>
<tr>
<td><code>sort(list)</code></td>
<td>Sorts a list into ascending order.</td>
</tr>
<tr>
<td><code>min(list)</code></td>
<td>Returns the smallest value in a list.</td>
</tr>
<tr>
<td><code>max(list)</code></td>
<td>Returns the largest value in a list.</td>
</tr>
<tr>
<td><code>reverse(list)</code></td>
<td>Reverses the order of elements in a list.</td>
</tr>
<tr>
<td><code>shuffle(list)</code></td>
<td>Randomly rearranges the elements in a list.</td>
</tr>
<tr>
<td><code>swap(list, p1, p2)</code></td>
<td>Exchanges the elements at index positions <code>p1</code> and <code>p2</code>.</td>
</tr>
<tr>
<td><code>replaceAll(list, x1, x2)</code></td>
<td>Replaces all elements matching <code>x1</code> with <code>x2</code>.</td>
</tr>
</tbody>
</table>

- The `java.util` package exports a similar Arrays class that provides the same basic operations for any array.