Chapter 20 ArrayLists

CS1: Java Programming
Colorado State University

Original slides by Daniel Liang
Modified slides by Chris Wilcox

Objectives

- To explore the relationship between interfaces and classes in the Java Collections Framework hierarchy (§20.2).
- To use the common methods defined in the **Collection** interface for operating collections (§20.2).
- To use the **Iterator** interface to traverse the elements in a collection (§20.3).
- To use a for-each loop to traverse the elements in a collection (§20.3).
- To explore how and when to use **ArrayList** or **LinkedList** to store elements (§20.4).
- To compare elements using the **Comparable** interface and the **Comparator** interface (§20.5).
- To use the static utility methods in the **Collections** class for sorting, searching, shuffling lists, and finding the largest and smallest element in collections (§20.6).
- To develop a multiple bouncing balls application using **ArrayList** (§20.7).

What is Data Structure?

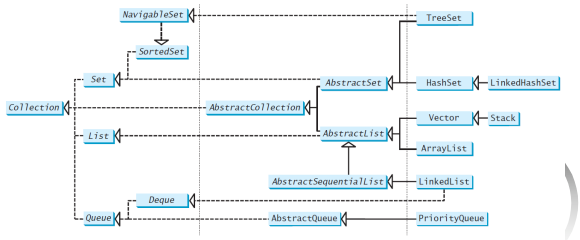
A data structure is a collection of data organized in some fashion. The structure not only stores data, but also supports operations for accessing and manipulating the data.

Java Collection Framework hierarchy

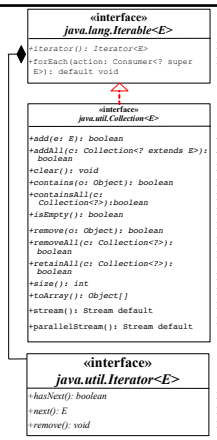
A *collection* is a container object that holds a group of objects, often referred to as *elements*. The Java Collections Framework supports three types of collections, named *lists, sets,* and *maps*.

Java Collection Framework hierarchy, cont.

Set and List are subinterfaces of Collection.



The Collection Interface



Returns an iterator for the elements in this collection.
Performs an action for each element in this iterator.

The Collection interface is for manipulating a collection of objects.

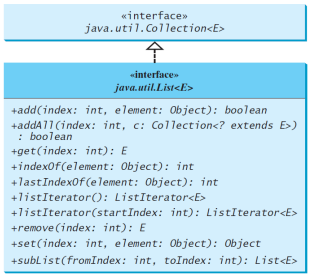
Adds a new element e to this collection.
Adds all the elements in the collection c to this collection.
Removes all the elements from this collection.
Returns true if this collection contains the element o.
Returns true if this collection contains all the elements in c.
Returns true if this collection contains no elements.
Removes the element o from this collection.
Removes all the elements in c from this collection.
Retains the elements that are both in c and in this collection.
Returns the number of elements in this collection.
Returns an array of Object for the elements in this collection.
Returns a stream from this collection (covered in Ch 23).
Returns a parallel stream from this collection (covered in Ch 23).

Returns true if this iterator has more elements to traverse.
Returns the next element from this iterator.
Removes the last element obtained using the next method.

The List Interface

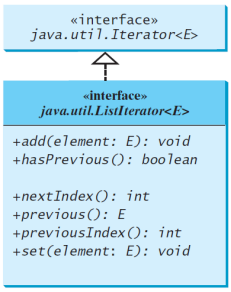
A list stores elements in a sequential order, and allows the user to specify where the element is stored. The user can access the elements by index.

The List Interface, cont.



Adds a new element at the specified index.
Adds all the elements in c to this list at the specified index.
Returns the element in this list at the specified index.
Returns the index of the first matching element.
Returns the index of the last matching element.
Returns the list iterator for the elements in this list.
Returns the iterator for the elements from `startIndex`.
Removes the element at the specified index.
Sets the element at the specified index.
Returns a sublist from `fromIndex` to `toIndex-1`.

The List Iterator

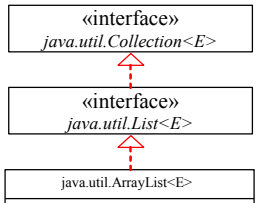


+add(element: E): void	Adds the specified object to the list.
+hasPrevious(): boolean	Returns true if this list iterator has more elements when traversing backward.
+nextIndex(): int	Returns the index of the next element.
+previous(): E	Returns the previous element in this list iterator.
+previousIndex(): int	Returns the index of the previous element.
+set(element: E): void	Replaces the last element returned by the previous or next method with the specified element.

ArrayList and LinkedList

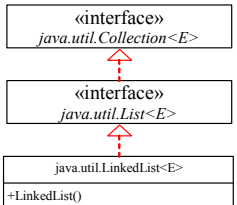
The ArrayList class and the LinkedList class are concrete implementations of the List interface. Which of the two classes you use depends on your specific needs. If you need to support random access through an index without inserting or removing elements from any place other than the end, ArrayList offers the most efficient collection. If, however, your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList. A list can grow or shrink dynamically. An array is fixed once it is created. If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

java.util.ArrayList



+ArrayList()	Creates an empty list with the default initial capacity.
+ArrayList(c: Collection<? extends E>)	Creates an array list from an existing collection.
+ArrayList(initialCapacity: int)	Creates an empty list with the specified initial capacity.
+trimToSize(): void	Trims the capacity of this ArrayList instance to be the list's current size.

java.util.LinkedList



+LinkedList()	Creates a default empty linked list.
+LinkedList(c: Collection<? extends E>)	Creates a linked list from an existing collection.
+addFirst(o: E): void	Adds the object to the head of this list.
+addLast(o: E): void	Adds the object to the tail of this list.
+getFirst(): E	Returns the first element from this list.
+getLast(): E	Returns the last element from this list.
+removeFirst(): E	Returns and removes the first element from this list.
+removeLast(): E	Returns and removes the last element from this list.

Example: Using ArrayList and LinkedList

This example creates an array list filled with numbers, and inserts new elements into the specified location in the list. The example also creates a linked list from the array list, inserts and removes the elements from the list. Finally, the example traverses the list forward and backward.

TestArrayAndLinkedList Run

The Comparator Interface

Sometimes you want to compare the elements of different types. The elements may not be instances of `Comparable` or are not comparable. You can define a comparator to compare these elements. To do so, define a class that implements the `java.util.Comparator` interface. The `Comparator` interface has the `compare` method for comparing two objects.

The Comparator Interface

public int compare(Object element1, Object element2)

Returns a negative value if element1 is less than element2, a positive value if element1 is greater than element2, and zero if they are equal.

GeometricObjectComparator

TestComparator Run

Other Comparator Examples

SortStringByLength Run

SortStringIgnoreCase Run

The Collections Class

The Collections class contains various static methods for operating on collections and maps, for creating synchronized collection classes, and for creating read-only collection classes.

The Collections Class UML Diagram