Chapter 20 Lists, Stacks, Queues, and Priority Queues

What is a Data Structure?

- A collection of data *elements*
- Stored in a structured fashion
- With operations that access & manipulate elements

Java Collections Framework

- *Collection* is a java interface
  - Java.utils.Container
- Defines abstract methods for objects that contain other objects (*elements*)
  - Add(E e)
  - Remove(E e)
  - Contains(E e)
  - toArray(E e)

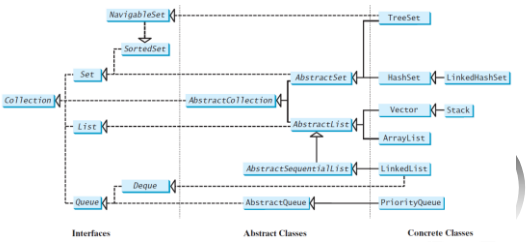
These are examples, not an exhaustive list

Three Types of Collections (interfaces that implement Collection)

- Lists – Stores elements in sequential order
  - Ordered Collection
- Sets – lists allow duplicates, sets do not
  - Unordered Collection
- Maps – data structure based on {key, value} pair
  - Holds two objects per entry
  - May contain duplicate values
  - Keys are always unique

Java Collections Framework

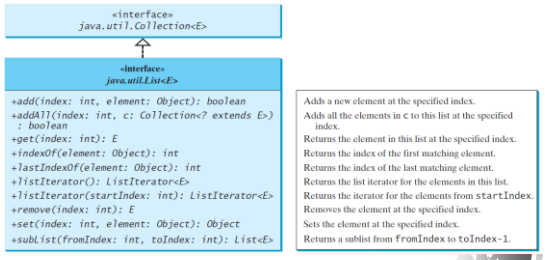
Set and List are subinterfaces of Collection.



The List Interface

- Elements stored in sequential order
- Programs can specify where an element is stored.
- Programs can access elements by index.

The List Interface, cont.

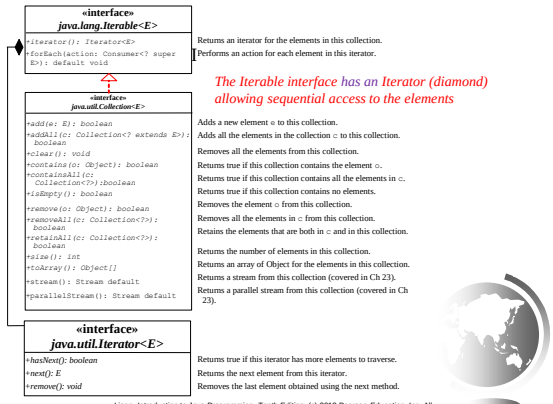


Iterators

- An iterator is a generalization of a reference
  - An abstract way of accessing an element
- *Iterator* is an interface
  - Java.util.Iterator
- Methods for sequentially accessing elements
  - hasNext()
  - next()
  - remove()

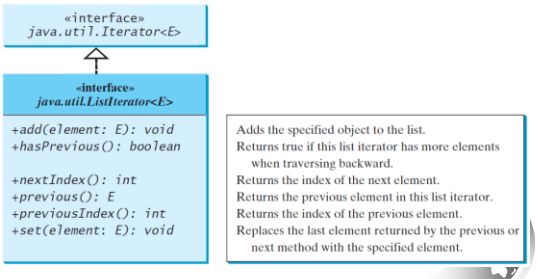
Why Iterators?

- Iterators allow you to abstract away the data structure
- Given an iterator, you can access elements in order
  - In a list
  - In a set
  - In a map
- The *Iterable* interface requires an object to implement iterators



The Iterable interface has an Iterator (diamond) allowing sequential access to the elements

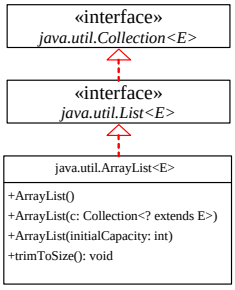
The List Iterator



Array vs ArrayList vs LinkedList

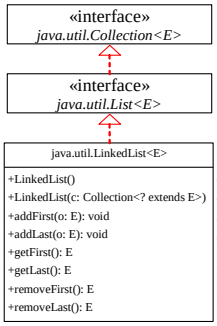
- ArrayList class and the LinkedList class
  - Concrete implementations of the List interface.
  - Usage depends on your specific needs.
- Efficiency
  - ArrayList – Fast random access through indices
  - LinkedList – Fast insertion and deletion of elements at specific locations
  - Array – Does not support insertion or deletion of elements
    - But the most efficient if insert/delete not needed

java.util.ArrayList

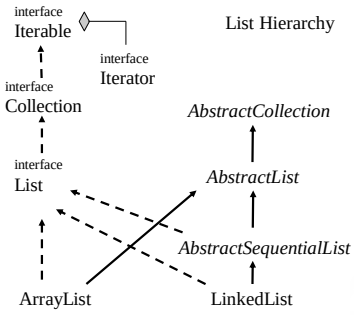


Creates an empty list with the default initial capacity.
Creates an array list from an existing collection.
Creates an empty list with the specified initial capacity.
Trims the capacity of this ArrayList instance to be the list's current size.

java.util.LinkedList



Creates a default empty linked list.
Creates a linked list from an existing collection.
Adds the object to the head of this list.
Adds the object to the tail of this list.
Returns the first element from this list.
Returns the last element from this list.
Returns and removes the first element from this list.
Returns and removes the last element from this list.



List Hierarchy

Example: Using ArrayList and LinkedList

- Create an array list filled with numbers
- Insert new elements in specific locations
- Create a linked list from the array list
- Insert and remove elements from the list.
- Traverse the list forward and backward.

TestArrayAndLinkedList Run

Comparable vs Comparator

- Comparable
  - Implemented with compareTo
  - Defines the natural order for the object
    - i.e. the order you will use most of the time
- Comparator
  - Implemented with compare()
  - Define an order for a specific purpose

The Comparator Interface

- An interface for comparing arbitrary elements
  - The elements don't have to be *Comparable*
  - Java.util.Comparator
- Defines a method called compare(T o1, T o2)
- Used as an argument to methods like sort(collection, CompareObject)

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.

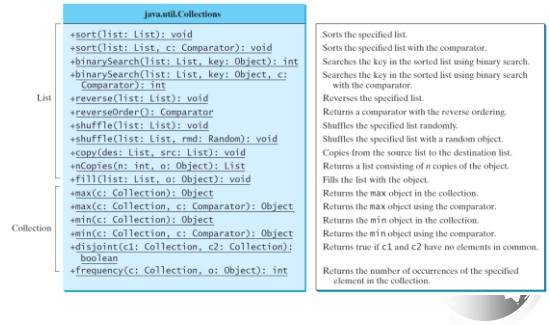
GeometricObjectComparator

TestComparator Run

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

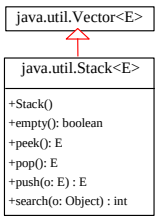


The Vector and Stack Classes

The Java Collections Framework was introduced with Java 2. Several data structures were supported prior to Java 2. Among them are the Vector class and the Stack class. These classes were redesigned to fit into the Java Collections Framework, but their old-style methods are retained for compatibility. This section introduces the Vector class and the Stack class.

The Stack Class

The vector class is deprecated, but similar to ArrayList



The Stack class represents a last-in-first-out stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element from the top of the stack.

Queues and Priority Queues

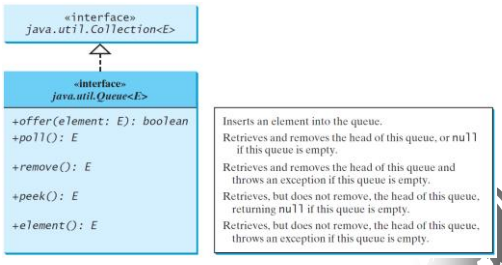
Queue is a first-in/first-out data structure.

- Elements are appended to the end of the queue.
- Elements are removed from the beginning of the queue.

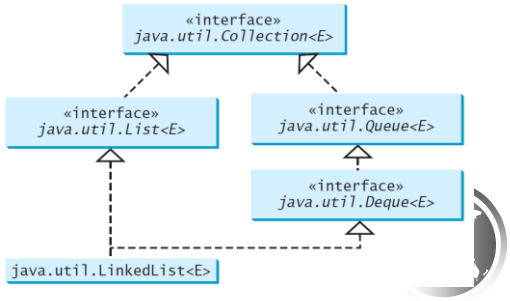
Priority queues assign priorities to elements.

- The element with the highest priority is removed first.

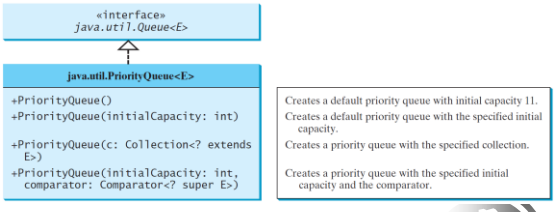
The Queue Interface



Using LinkedList for Queue



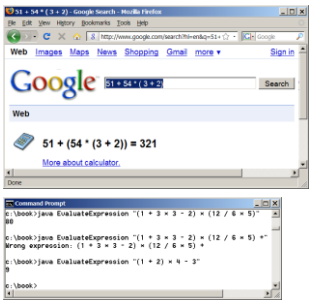
The PriorityQueue Class



PriorityQueueDemo Run

Case Study: Evaluating Expressions

Stacks can be used to evaluate expressions.



Evaluate Expression Run

Some examples

- 2 + 3

When we see + we haven't seen operand 3 yet. Use an operandStack to push operands, and an operatorStack to push operators:

push (2, operandStack)

push (+, operatorStack)

push (3, operandStack)

End of expression: apply operator to operands

Why wait until we see the end or rest of expression?

2+3*4

- 2 + 3 – 4 is (2+3) – 4, and NOT 2 + (3-4)

push (2, operandStack)

push (+, operatorStack)

push (3, operandStack)

Seeing -: apply operator on stack to operands

push(-, operatorStack)

push(4, operandStack)

End: apply operator(s) to operands

- 2+3*4-5

push (2, operandStack)
push (+, operatorStack)
push (3, operandStack)
*: has precedence over +, so
push (*, operatorStack)
push (4, operandStack)
-: apply operators to operands,
push (-, operatorStack)
5:push (5, operandStack)
End: apply operators to operands

- 2*(3+4)/5

push (2, operandStack)
push (*, operatorStack)
(: make a substack at top of operatorStack:
push ('(', operatorStack)
push (3, operandStack)
push (+, operatorStack)
push (4, operandStack)
): apply operators to operands until '(', pop ('(')
push (/, operatorStack)
push (5, operandStack)
End: apply operators to operands

Algorithm

Phase 1: Scanning the expression

The program scans the expression from left to right to extract operands, operators, and the parentheses.

1.1. If the extracted item is an operand, push it to **operandStack**.

1.2. If the extracted item is a **+** or **-** operator, process all the operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.3. If the extracted item is a ***** or **/** operator, process the ***** or **/** operators at the top of **operatorStack** and push the extracted operator to **operatorStack**.

1.4. If the extracted item is a **(** symbol, push it to **operatorStack**.

1.5. If the extracted item is a **)** symbol, repeatedly process the operators from the top of **operatorStack** until seeing the **(** symbol on the stack.

Phase 2: Clearing the stack

Repeatedly process the operators from the top of **operatorStack** until **operatorStack** is empty.

Example

Expression	*Scan*	*Action*	operandStack	operatorStack
(1 + 2)*4 − 3	(	Phase 1.4		(
(1 + 2)*4 − 3	1	Phase 1.1	1	(
(1 + 2)*4 − 3	+	Phase 1.2	1	+ (
(1 + 2)*4 − 3	2	Phase 1.1	2 1	(
(1 + 2)*4 − 3	)	Phase 1.5	3	
(1 + 2)*4 − 3	*	Phase 1.3	3	*
(1 + 2)*4 − 3	4	Phase 1.1	4 3	*
(1 + 2)*4 − 3	−	Phase 1.2	12	−
(1 + 2)*4 − 3	3	Phase 1.1	3 12	−
(1 + 2)*4 − 3	none	Phase 2	9	

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).
- To distinguish between **Vector** and **ArrayList** and to use the **Stack** class for creating stacks (§20.8).
- To explore the relationships among **Collection**, **Queue**, **LinkedList**, and **PriorityQueue** and to create priority queues using the **PriorityQueue** class (§20.9).
- To use stacks to write a program to evaluate expressions (§20.10).