Chapter 20 Lists, Stacks, Queues, and Priority Queues

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).

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 Collections Framework

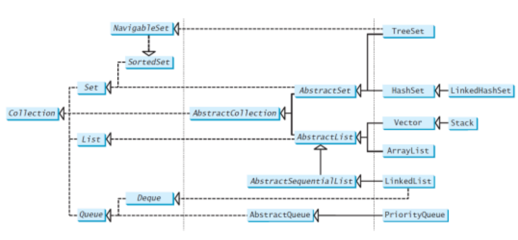
A *collection* is a container object that holds a group of objects, often referred to as *elements*. The Java Collections Framework supports three types of collections, named *lists, sets,* and *maps*.

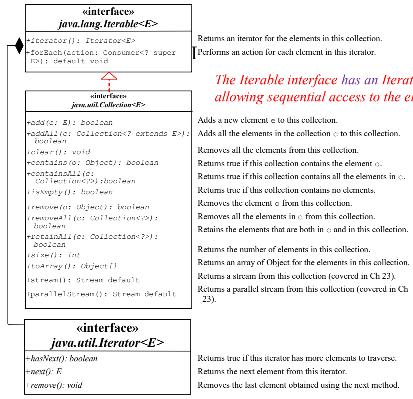
Java Collections Framework

- 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.

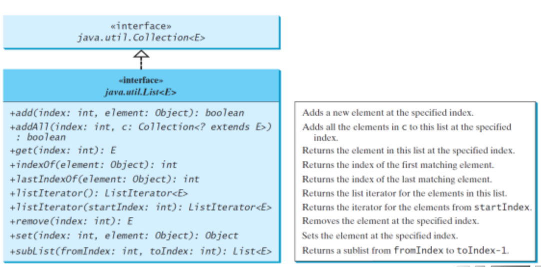




The List Interface

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.



The List Iterator

«interface» java.util.Iterator<E>

«interface» java.util.ListIterator<E>	
+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.

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 – Random access through an index
  - LinkedList - Insertion or deletion of elements at any location
  - Array - If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

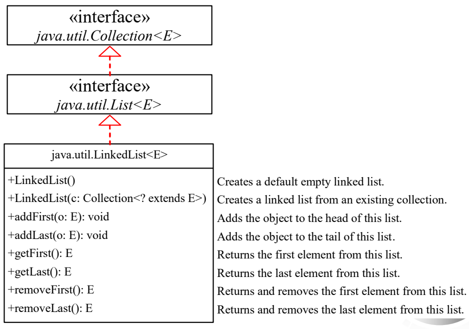
java.util.ArrayList

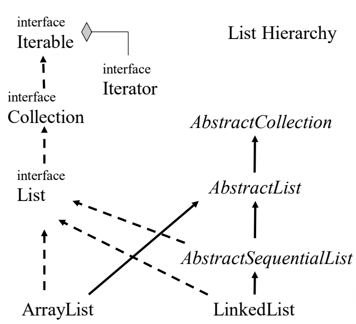
«interface» java.util.Collection<E>

«interface» java.util.List<E>

java.util.ArrayList<E>	
+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





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

Comparable vs Comparator

- Comparable
  - Implemented with compareTo
  - Defines the natural order for the object
- Comparator
  - Implemented with compare()
  - Defines a different order for some purpose

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

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

	java.util.Collections	
List	+sort(list: List): void	Sorts the specified list.
	+sort(list: List, c: Comparator): void	Sorts the specified list with the comparator.
	+binarySearch(list: List, key: Object): int	Searches the key in the sorted list using binary search.
	+binarySearch(list: List, key: Object, c: Comparator): int	Searches the key in the sorted list using binary search with the comparator.
	+reverse(list: List): void	Reverses the specified list.
	+reverseOrder(): Comparator	Returns a comparator with the reverse ordering.
	+shuffle(list: List): void	Shuffles the specified list randomly.
	+shuffle(list: List, rmd: Random): void	Shuffles the specified list with a random object.
	+copy(des: List, src: List): void	Copies from the source list to the destination list.
	+nCopies(n: int, o: Object): List	Returns a list consisting of *n* copies of the object.
	+fill(list: List, o: Object): void	Fills the list with the object.
Collection	+max(c: Collection): Object	Returns the max object in the collection.
	+max(c: Collection, c: Comparator): Object	Returns the max object using the comparator.
	+min(c: Collection): Object	Returns the min object in the collection.
	+min(c: Collection, c: Comparator): Object	Returns the min object using the comparator.
	+disjoint(c1: Collection, c2: Collection): boolean	Returns true if c1 and c2 have no elements in common.
	+frequency(c: Collection, o: Object): int	Returns the number of occurrences of the specified element in the collection.

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 Vector Class

In Java 2, Vector is the same as ArrayList, except that Vector contains the *synchronized* methods for accessing and modifying the vector. None of the new collection data structures introduced so far are synchronized. If synchronization is required, you can use the synchronized versions of the collection classes. These classes are introduced later in the section, “The Collections Class.”

The Vector Class, cont.

java.util.AbstractList<E>

↑

java.util.Vector<E>	
+Vector()	Creates a default empty vector with initial capacity 10.
+Vector(c: Collection<? extends E>)	Creates a vector from an existing collection.
+Vector(initialCapacity: int)	Creates a vector with the specified initial capacity.
+Vector(initCapacity: int, capacityIncr: int)	Creates a vector with the specified initial capacity and increment.
+addElement(o: E): void	Appends the element to the end of this vector.
+capacity(): int	Returns the current capacity of this vector.
+copyInto(anArray: Object[]): void	Copies the elements in this vector to the array.
+elementAt(index: int): E	Returns the object at the specified index.
+elements(): Enumeration<E>	Returns an enumeration of this vector.
+ensureCapacity(): void	Increases the capacity of this vector.
+firstElement(): E	Returns the first element in this vector.
+insertElementAt(o: E, index: int): void	Inserts o into this vector at the specified index.
+lastElement(): E	Returns the last element in this vector.
+removeAllElements(): void	Removes all the elements in this vector.
+removeElement(o: Object): boolean	Removes the first matching element in this vector.
+removeElementAt(index: int): void	Removes the element at the specified index.
+setElementAt(o: E, index: int): void	Sets a new element at the specified index.
+setSize(newSize: int): void	Sets a new size in this vector.
+trimToSize(): void	Trims the capacity of this vector to its size.

The Stack Class

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.

java.util.Vector<E>

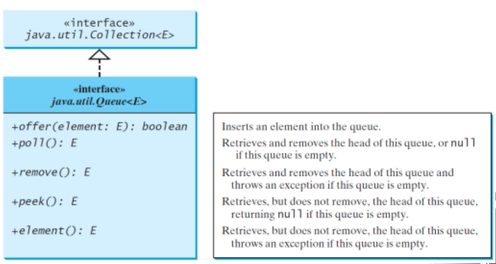
↑

java.util.Stack<E>	
+Stack()	Creates an empty stack.
+empty(): boolean	Returns true if this stack is empty.
+peek(): E	Returns the top element in this stack.
+pop(): E	Returns and removes the top element in this stack.
+push(o: E) : E	Adds a new element to the top of this stack.
+search(o: Object) : int	Returns the position of the specified element in this stack.

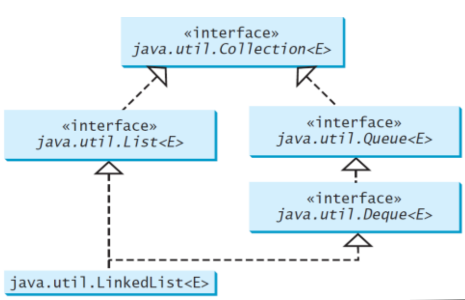
Queues and Priority Queues

A queue is a first-in/first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a priority queue, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first.

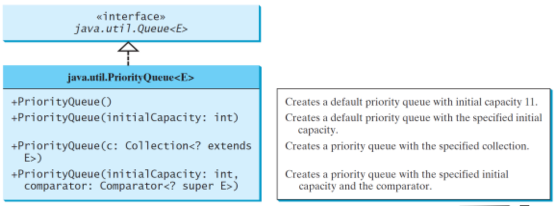
The Queue Interface



Using LinkedList for Queue



The PriorityQueue Class

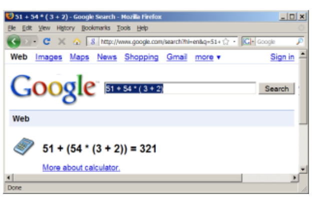


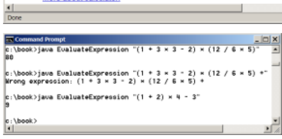
PriorityQueueDemo

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	