CSE 102 - COMPUTER PROGRAMMING II
DATA STRUCTURES

Joseph LEDET
Department of Computer Engineering
Akdeniz University
josephledet@akdeniz.edu.tr

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).

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*.

- **Class Container**
- java.lang.Object
  - java.awt.Component
  - java.awt.Container

public class **Container**

Extends Component

A generic Abstract Window Toolkit(AWT) container object is a component that can contain other AWT components.

Components added to a container are tracked in a list. The order of the list will define the components' front-to-back stacking order within the container.

If no index is specified when adding a component to a container, it will be added to the end of the list (and hence to the bottom of the stacking order).

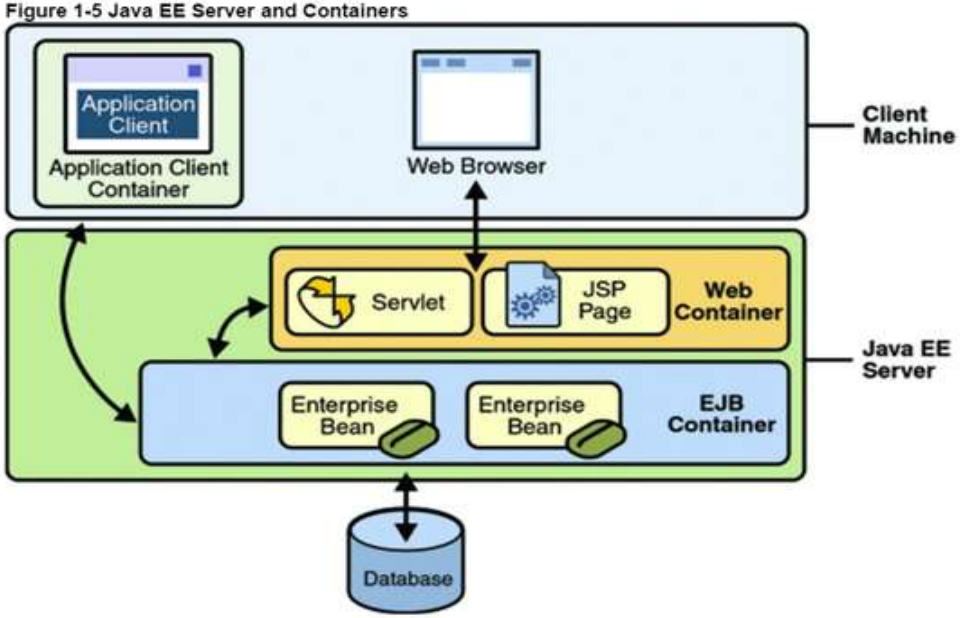
CONTAINERS

- **Containers** are the interface between a component and the low-level platform-specific functionality that supports the component.

Container Types

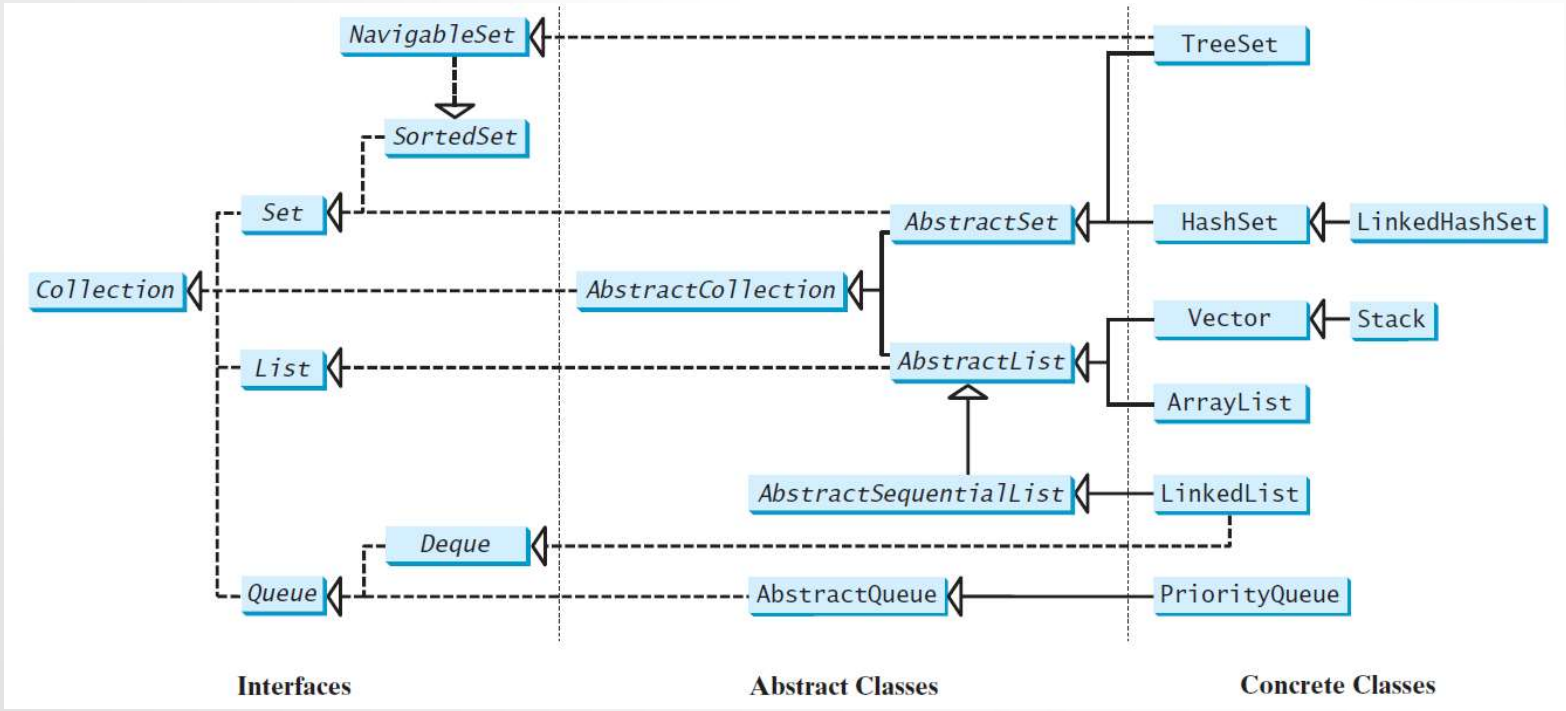
The deployment process installs Java EE application components in the Java EE containers as illustrated in Figure 1-5.

Figure 1-5 Java EE Server and Containers

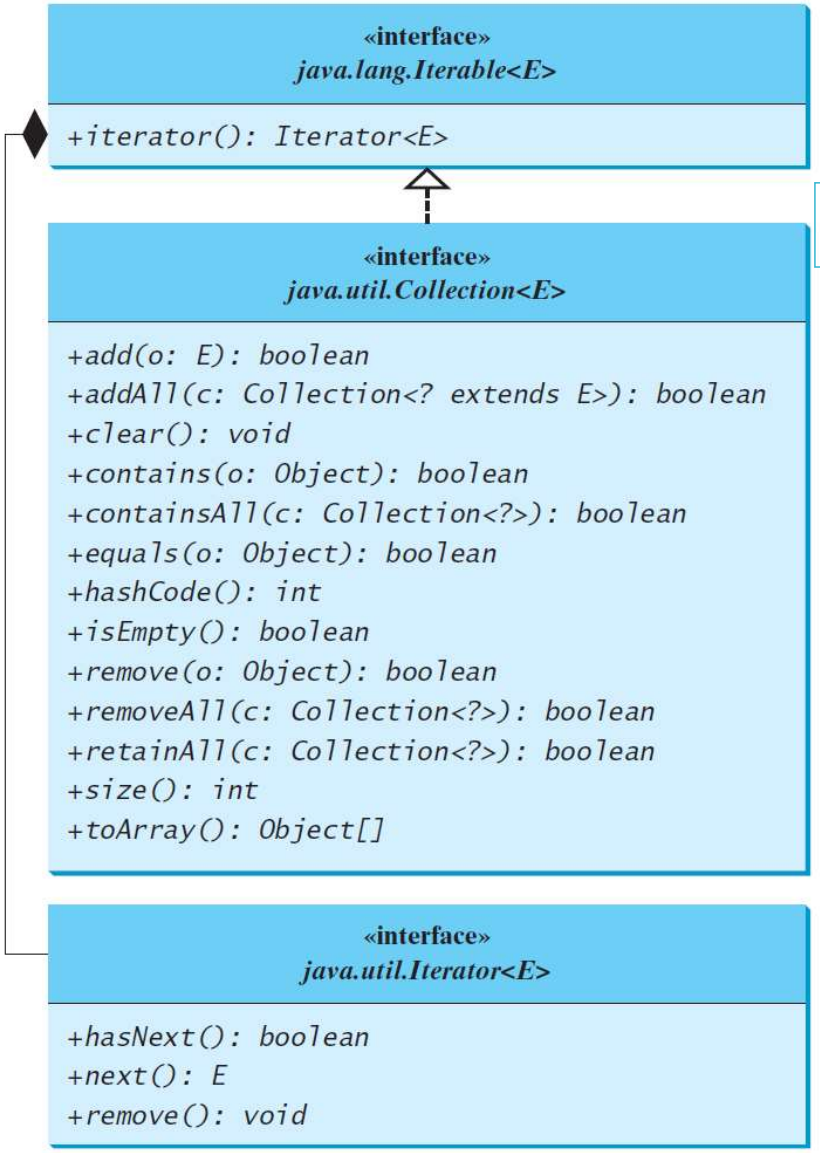


JAVA COLLECTION FRAMEWORK HIERARCHY, CONT.

- Set and List are subinterfaces of Collection.



THE COLLECTION INTERFACE



Returns an iterator for the elements in this collection.

The Collection interface is the root interface for manipulating a collection of objects.

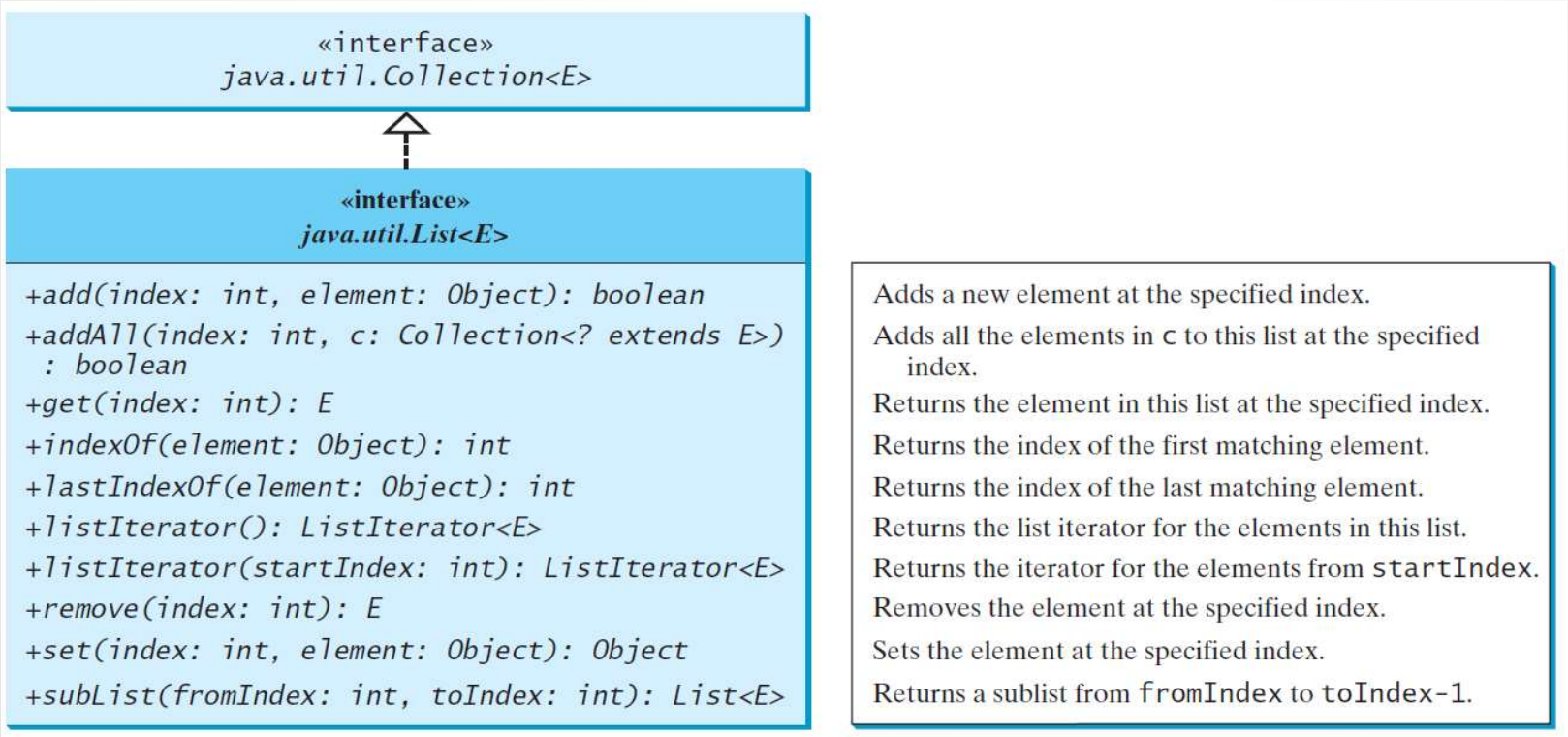
Adds a new element `o` to this collection.
Adds all the elements in the collection `c` to this collection.
Removes all the elements from this collection.
Returns true if this collection contains the element `o`.
Returns true if this collection contains all the elements in `c`.
Returns true if this collection is equal to another collection `o`.
Returns the hash code for this collection.
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 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.

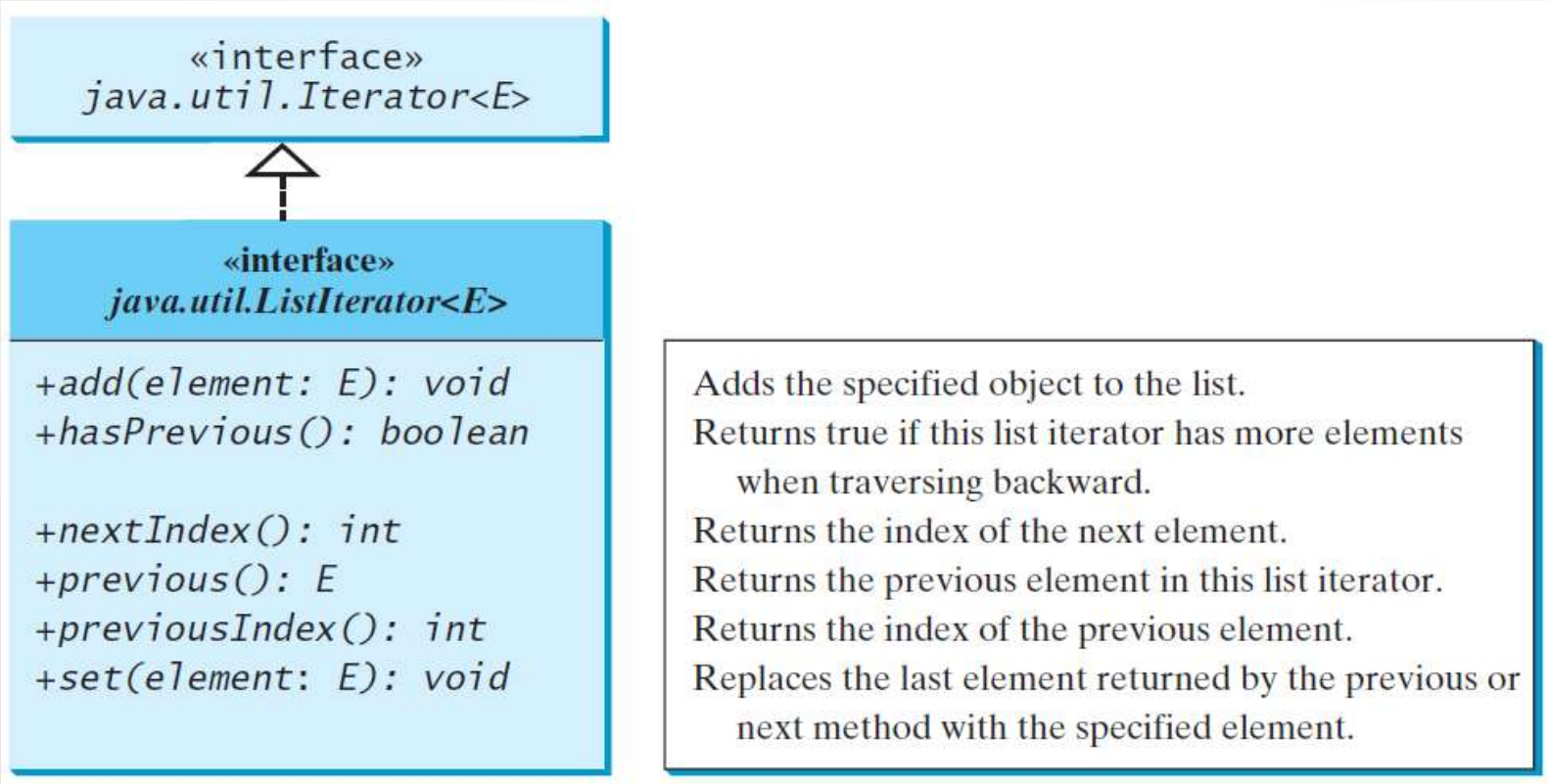
THE LIST INTERFACE

- 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.



THE LIST ITERATOR

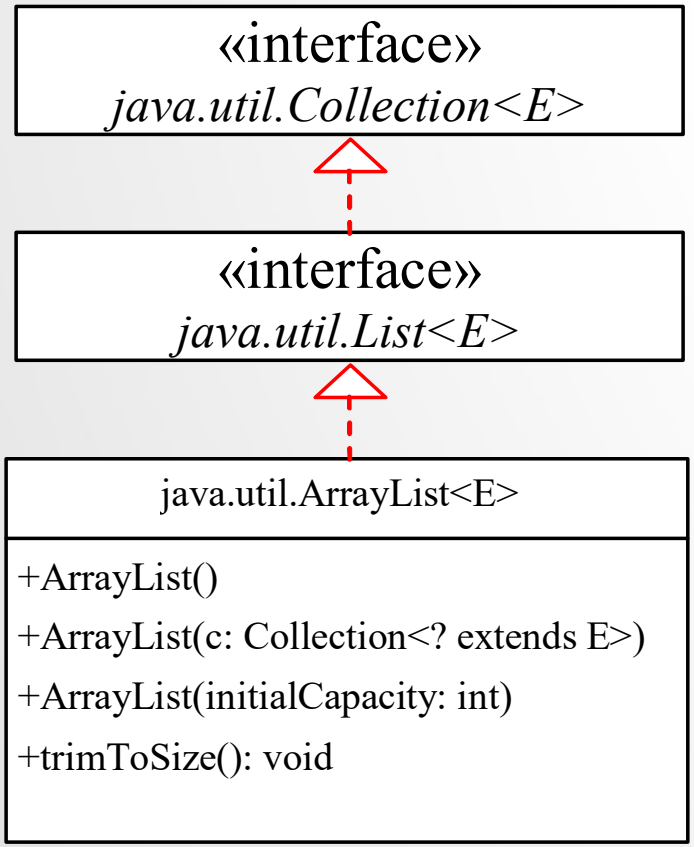


ARRAYLIST AND LINKEDLIST

- 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



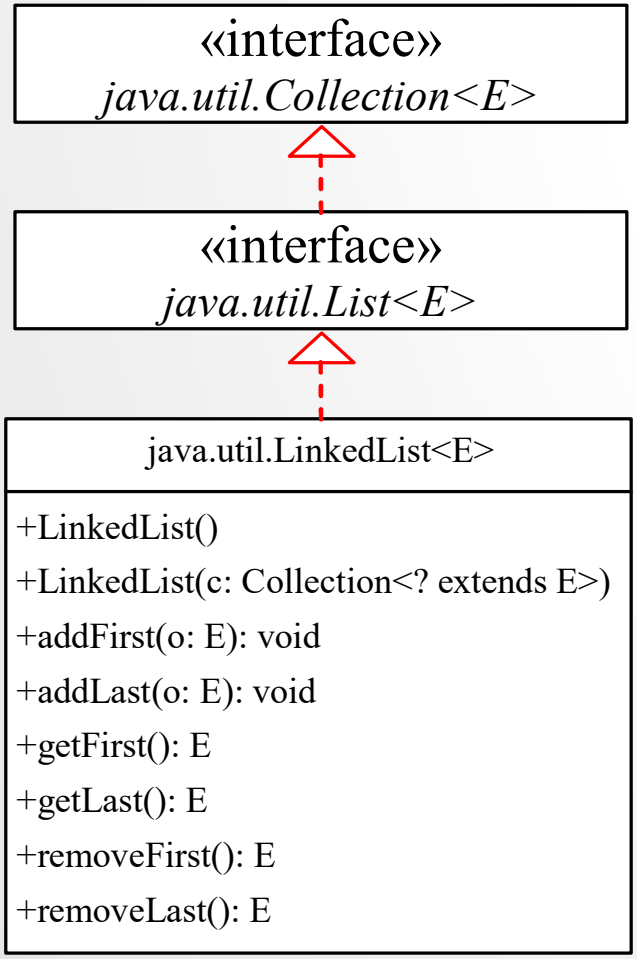
Creates an empty list with the default initial capacity.

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.

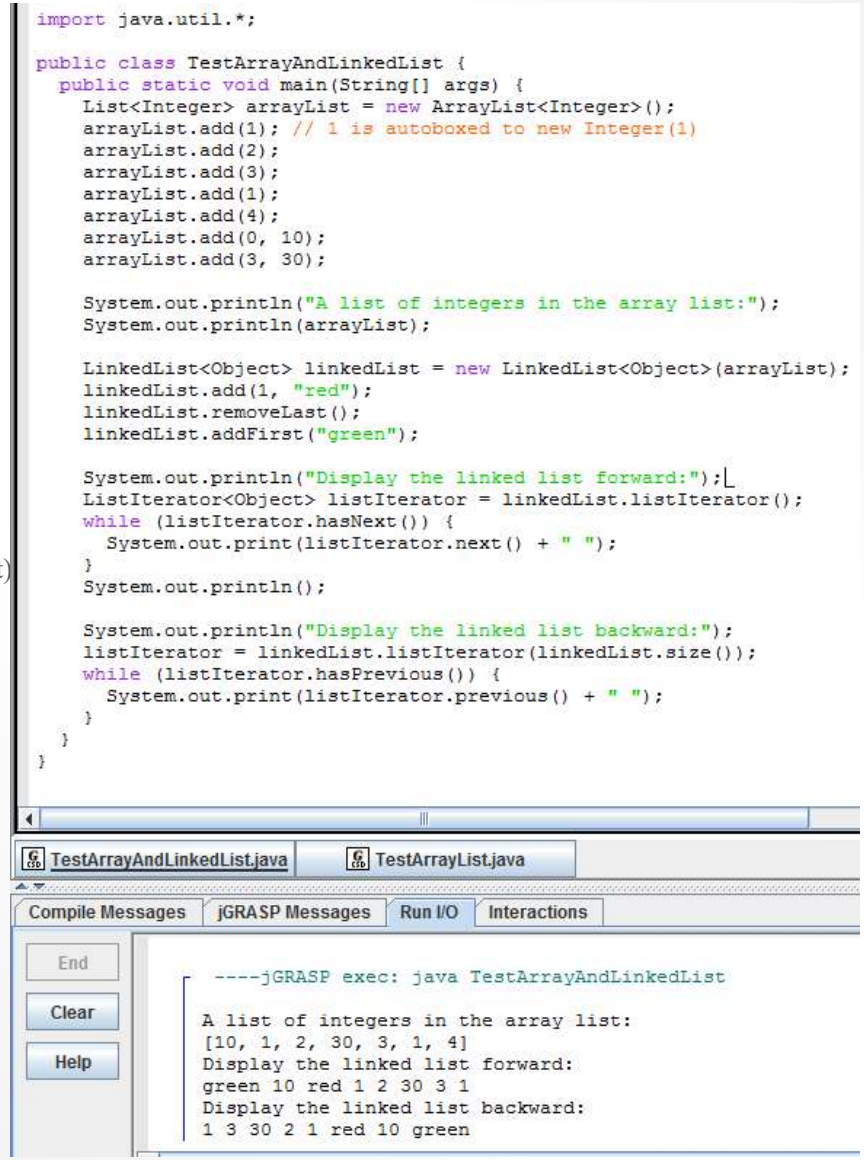
Returns and removes the last element from this list.

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

```
import java.util.*;

public class TestArrayAndLinkedList {
  public static void main(String[] args) {
    List<Integer> arrayList = new ArrayList<Integer>();
    arrayList.add(1); // 1 is autoboxed to new Integer(1)
    arrayList.add(2);
    arrayList.add(3);
    arrayList.add(1);
    arrayList.add(4);
    arrayList.add(0, 10);
    arrayList.add(3, 30);

    System.out.println("A list of integers in the array list:");
    System.out.println(arrayList);

    LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
    linkedList.add(1, "red");
    linkedList.removeLast();
    linkedList.addFirst("green");

    System.out.println("Display the linked list forward:");
    ListIterator<Object> listIterator = linkedList.listIterator();
    while (listIterator.hasNext()) {
      System.out.print(listIterator.next() + " ");
    }
    System.out.println();

    System.out.println("Display the linked list backward:");
    listIterator = linkedList.listIterator(linkedList.size());
    while (listIterator.hasPrevious()) {
      System.out.print(listIterator.previous() + " ");
    }
  }
}
```



```
public class TestArrayList {
  public static void main(String[] args) {
    // Create a list to store cities
    java.util.ArrayList cityList = new java.util.ArrayList();

    // Add some cities in the list
    cityList.add("London");
    // cityList now contains [London]
    cityList.add("Denver");
    // cityList now contains [London, Denver]
    cityList.add("Paris");
    // cityList now contains [London, Denver, Paris]
    cityList.add("Miami");
    // cityList now contains [London, Denver, Paris, Miami]
    cityList.add("Seoul");
    // contains [London, Denver, Paris, Miami, Seoul]
    cityList.add("Tokyo");
    // contains [London, Denver, Paris, Miami, Seoul, Tokyo]

    System.out.println("List size? " + cityList.size());
    System.out.println("Is Miami in the list? " +
      cityList.contains("Miami"));
    System.out.println("The location of Denver in the list? "
      + cityList.indexOf("Denver"));
    System.out.println("Is the list empty? " +
     cityList.isEmpty()); // Print false

    // Insert a new city at index 2
    cityList.add(2, "Xian");
    // contains [London, Denver, Xian, Paris, Miami, Seoul, Tokyo]

    // Remove a city from the list
    cityList.remove("Miami");
    // contains [London, Denver, Xian, Paris, Seoul, Tokyo]

    // Remove a city at index 1
    cityList.remove(1);
    // contains [London, Xian, Paris, Seoul, Tokyo]

    // Display the contents in the list
    System.out.println(cityList.toString());

    // Display the contents in the list in reverse order
    for (int i = cityList.size() - 1; i >= 0; i--)
      System.out.print(cityList.get(i) + " ");
    System.out.println();

    // Create a list to store two circles
    java.util.ArrayList list = new java.util.ArrayList();

    // Add two circles
    list.add(new Circle4(2));
    list.add(new Circle4(3));

    // Display the area of the first circle in the list
    System.out.println("The area of the circle? " +
      ((Circle4)list.get(0)).getArea());
  }
}
```

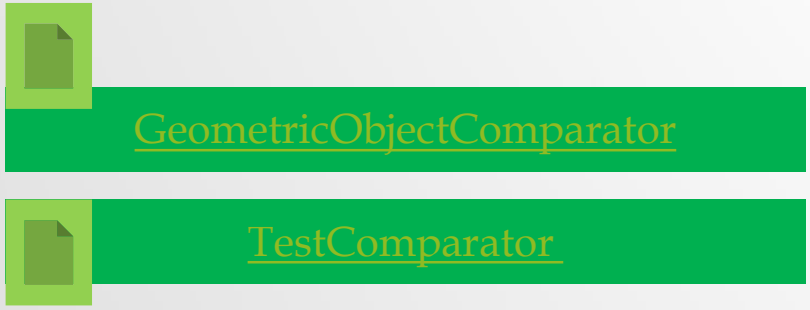
```
----jGRASP exec: java TestArrayList

List size? 6
Is Miami in the list? true
The location of Denver in the list? 1
Is the list empty? false
[London, Xian, Paris, Seoul, Tokyo]
Tokyo Seoul Paris Xian London
The area of the circle? 12.566370614359172

----jGRASP: operation complete.
```

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 two methods, compare and equals.

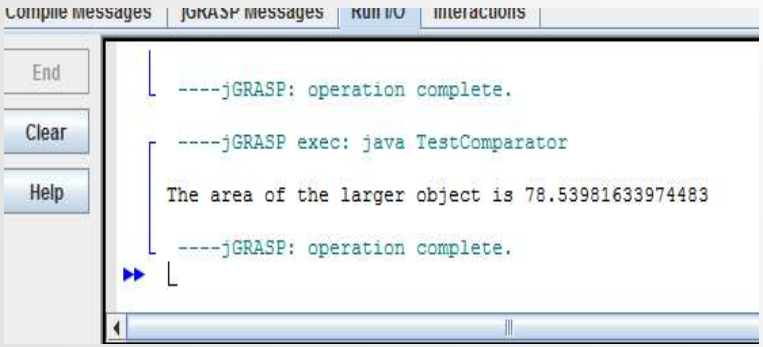
THE COMPARATOR INTERFACE

- 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.



Run

```
import java.util.Comparator;

public class GeometricObjectComparator
    implements
Comparator<GeometricObject>,
java.io.Serializable {
  public int compare(GeometricObject
o1, GeometricObject o2) {
    double area1 = o1.getArea();
    double area2 = o2.getArea();

    if (area1 < area2)
      return -1;
    else if (area1 == area2)
      return 0;
   else
      return 1;
  }
}
```

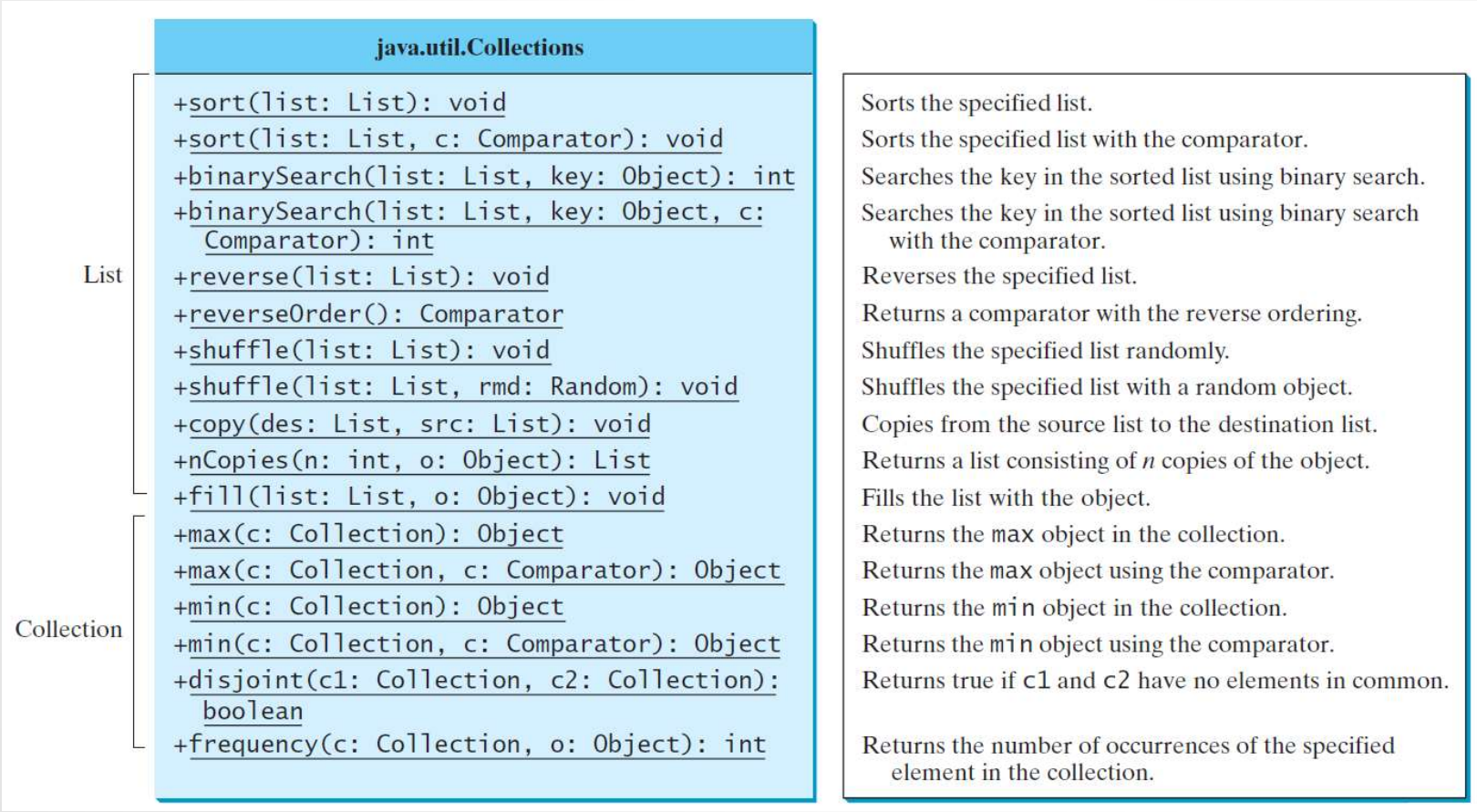



```
import java.util.Comparator;

public class TestComparator {
  public static void main(String[] args) {
    GeometricObject g1 = new Rectangle(5, 5);
    GeometricObject g2 = new Circle(5);

    GeometricObject g =
      max(g1, g2, new
GeometricObjectComparator());

    System.out.println("The area of the larger
object is " +
      g.getArea());
  }

  public static GeometricObject
max(GeometricObject g1,
      GeometricObject g2,
Comparator<GeometricObject> c) {
    if (c.compare(g1, g2) > 0)
      return g1;
    else
      return g2;
  }
}
```

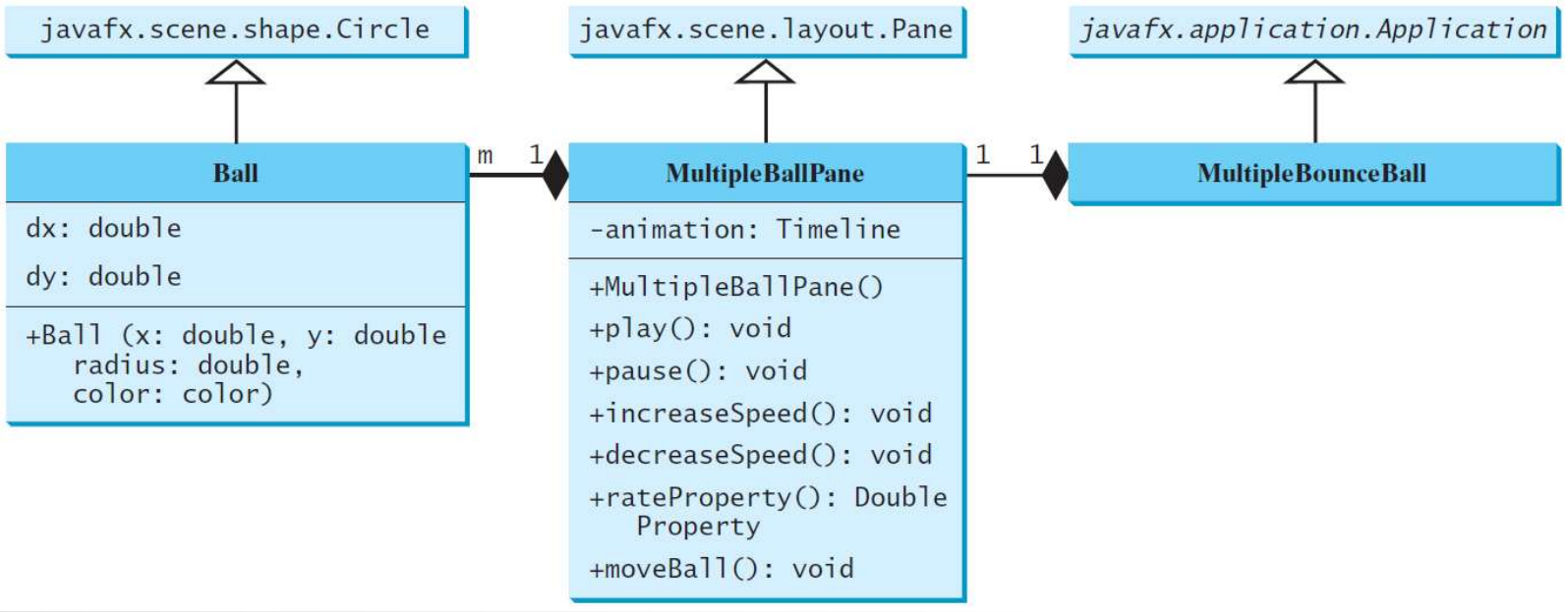
THE COLLECTIONS CLASS

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



CASE STUDY: MULTIPLE BOUNCING BALLS



MultipleBounceBall

Run

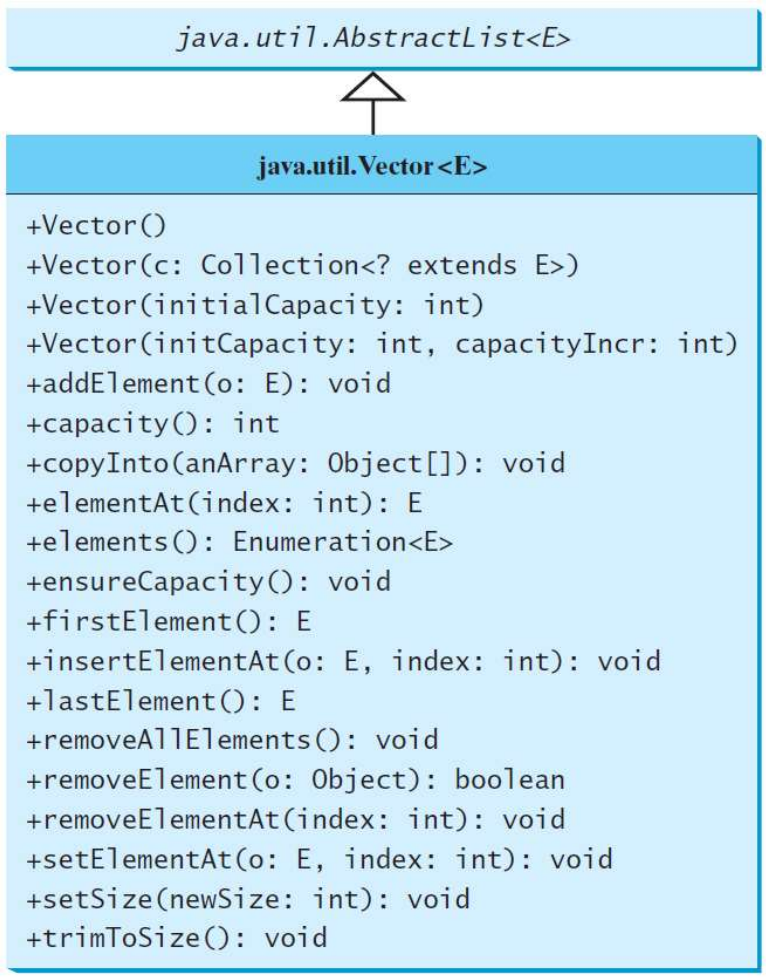
THE VECTOR AND STACK CLASSES

- 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.

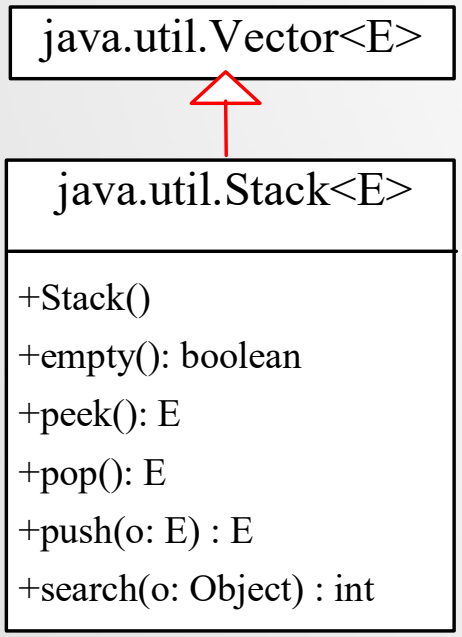


Creates a default empty vector with initial capacity 10.	
Creates a vector from an existing collection.	
Creates a vector with the specified initial capacity.	
Creates a vector with the specified initial capacity and increment.	
Appends the element to the end of this vector.	
Returns the current capacity of this vector.	
Copies the elements in this vector to the array.	
Returns the object at the specified index.	
Returns an enumeration of this vector.	
Increases the capacity of this vector.	
Returns the first element in this vector.	
Inserts o into this vector at the specified index.	
Returns the last element in this vector.	
Removes all the elements in this vector.	
Removes the first matching element in this vector.	
Removes the element at the specified index.	
Sets a new element at the specified index.	
Sets a new size in this vector.	
Trims the capacity of this vector to its size.	

THE STACK CLASS

The Stack class represents a last-in-first-out (LIFO)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.



Creates an empty stack.

Returns true if this stack is empty.

Returns the top element in this stack.

Returns and removes the top element in this stack.

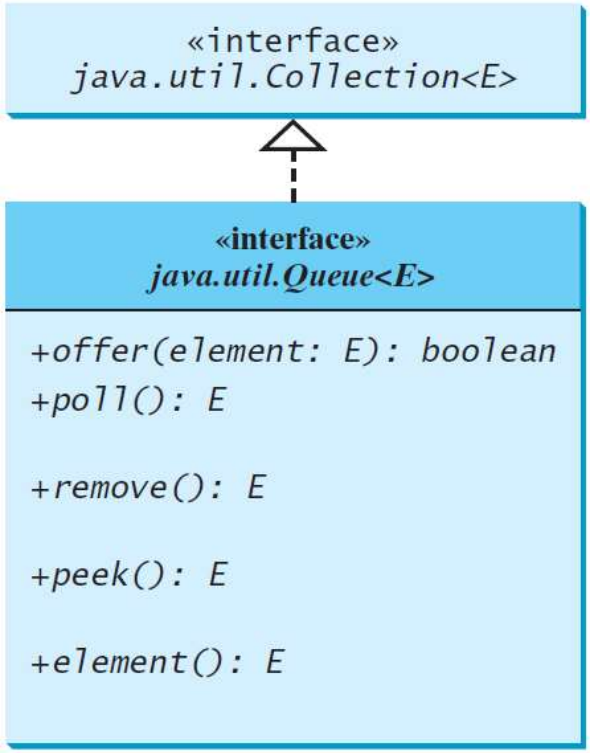
Adds a new element to the top of this stack.

Returns the position of the specified element in this stack.

QUEUES AND PRIORITY QUEUES

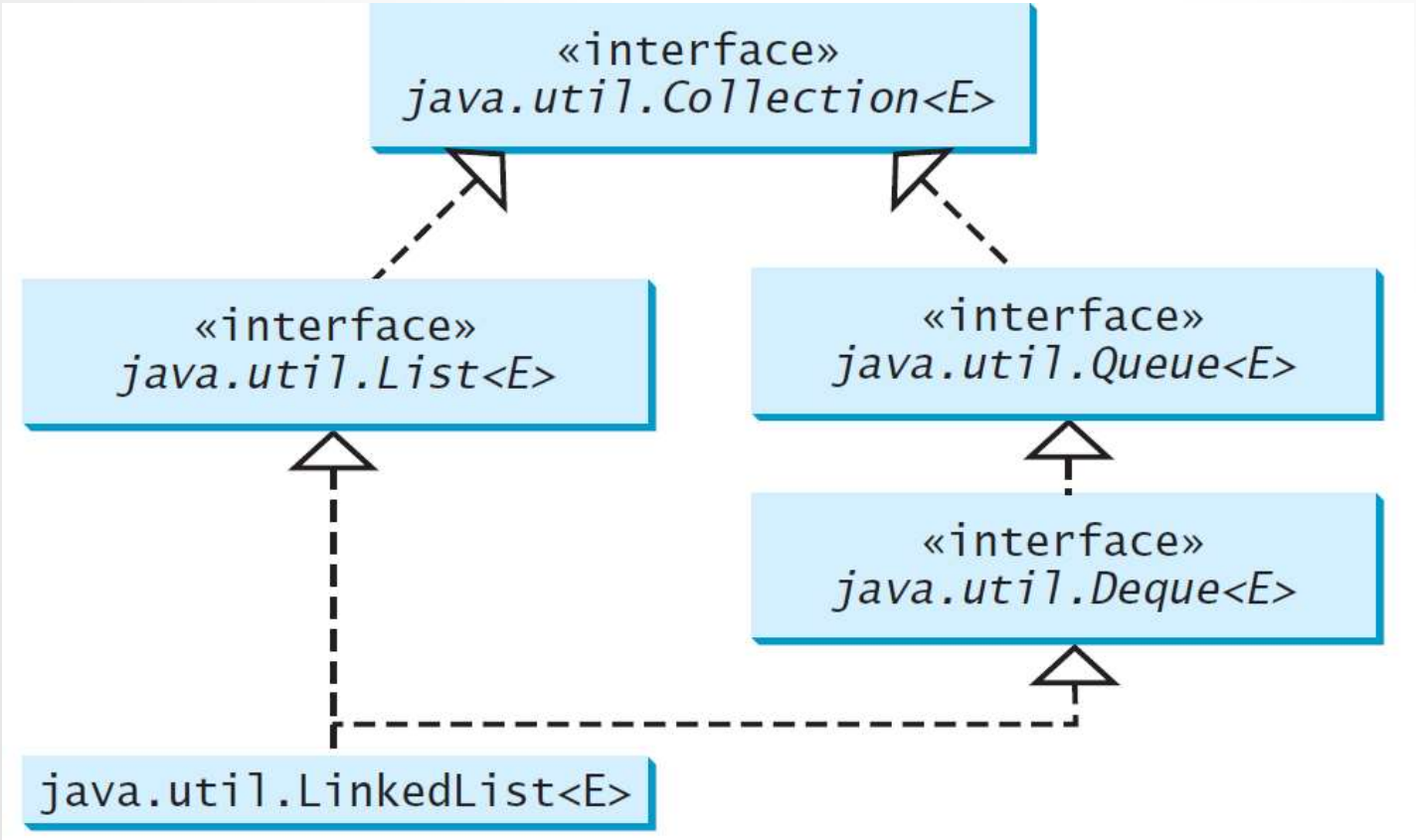
- A queue is a first-in/first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a priority queue, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first.

THE QUEUE INTERFACE

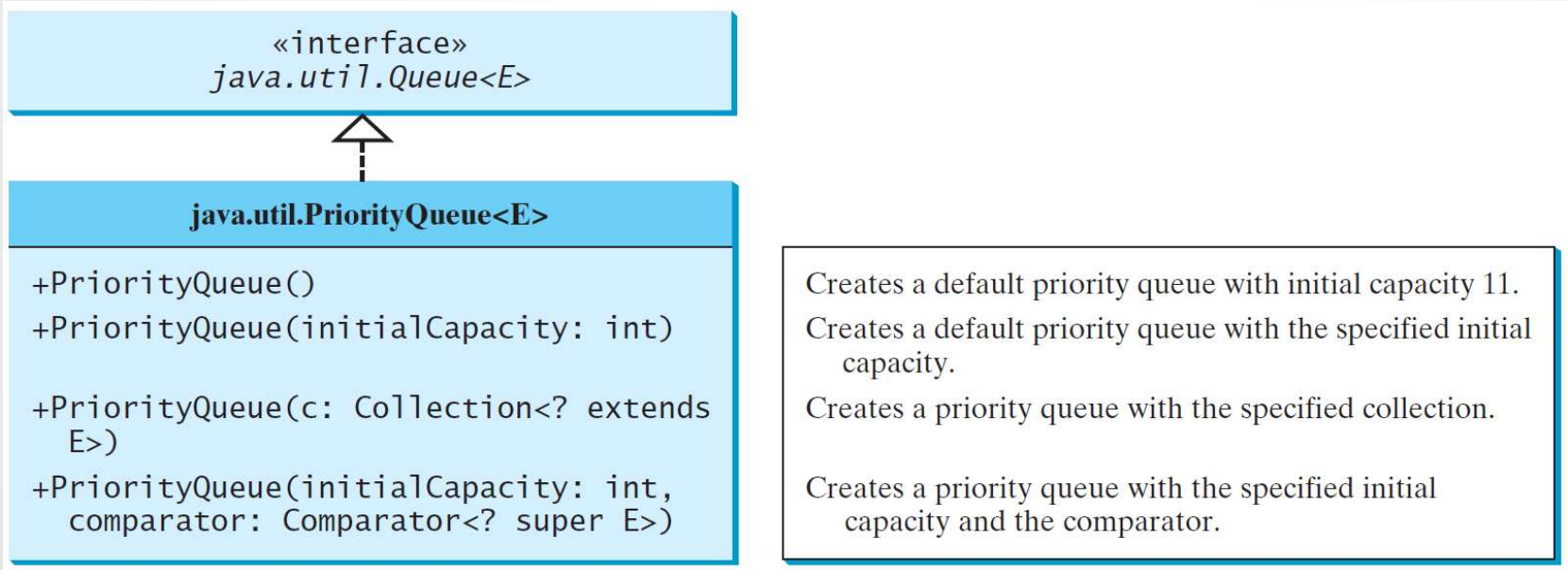


+offer(element: E): boolean	Inserts an element into the queue.
+poll(): E	Retrieves and removes the head of this queue, or `null` if this queue is empty.
+remove(): E	Retrieves and removes the head of this queue and throws an exception if this queue is empty.
+peek(): E	Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
+element(): E	Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.

USING LINKEDLIST FOR QUEUE



THE PRIORITYQUEUE CLASS



PriorityQueueDemo

Run

```
import java.util.*;

public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<String> queue1 = new
PriorityQueue<String>();
    queue1.offer("Oklahoma");
    queue1.offer("Indiana");
    queue1.offer("Georgia");
    queue1.offer("Texas");

    System.out.println("Priority queue using Comparable:");
    while (queue1.size() > 0) {
      System.out.print(queue1.remove() + " ");
    }

    PriorityQueue<String> queue2 = new
PriorityQueue<String>(
      4, Collections.reverseOrder());
    queue2.offer("Oklahoma");
    queue2.offer("Indiana");
    queue2.offer("Georgia");
    queue2.offer("Texas");

    System.out.println("\nPriority queue using Comparator:");
    while (queue2.size() > 0) {
      System.out.print(queue2.remove() + " ");
    }
  }
}
```

```
import java.util.*;

public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<String> queue1 = new PriorityQueue<String>();
    queue1.offer("Oklahoma");
    queue1.offer("Indiana");
    queue1.offer("Georgia");
    queue1.offer("Texas");

    System.out.println("Priority queue using Comparable:");
    while (queue1.size() > 0) {
      System.out.print(queue1.remove() + " ");
    }

    PriorityQueue<String> queue2 = new PriorityQueue<String>(
      4, Collections.reverseOrder());
    queue2.offer("Oklahoma");
    queue2.offer("Indiana");
    queue2.offer("Georgia");
    queue2.offer("Texas");

    System.out.println("\nPriority queue using Comparator:");
    while (queue2.size() > 0) {
      System.out.print(queue2.remove() + " ");
    }
  }
}
```

GeometricObjectComparator.java | PriorityQueueDemo.java

Compile Messages | jGRASP Messages | Run I/O | Interactions

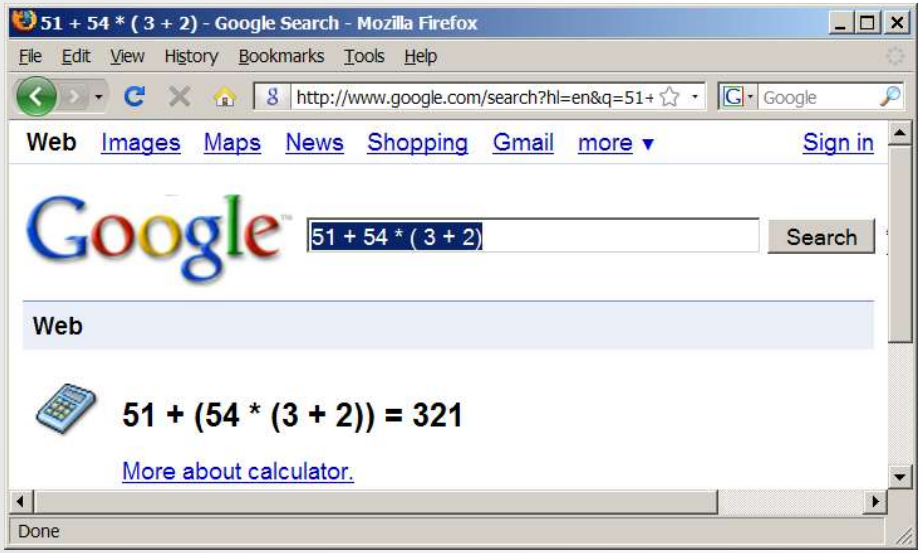
Clear

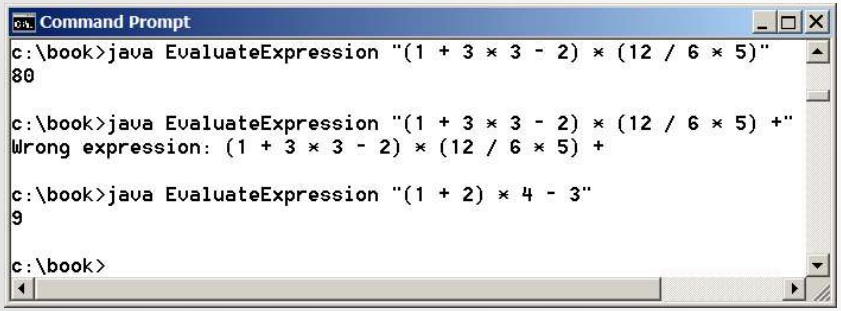
Help

```
 ----jGRASP exec: java PriorityQueueDemo

Priority queue using Comparable:
Georgia Indiana Oklahoma Texas
Priority queue using Comparator:
Texas Oklahoma Indiana Georgia
 ----jGRASP: operation complete.
```

CASE STUDY: EVALUATING EXPRESSIONS

Stacks can be used to evaluate expressions.





Evaluate Expression

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	