Data Structures
Lists, Stacks, Queues, and Priority Queues
(Chapter 20)

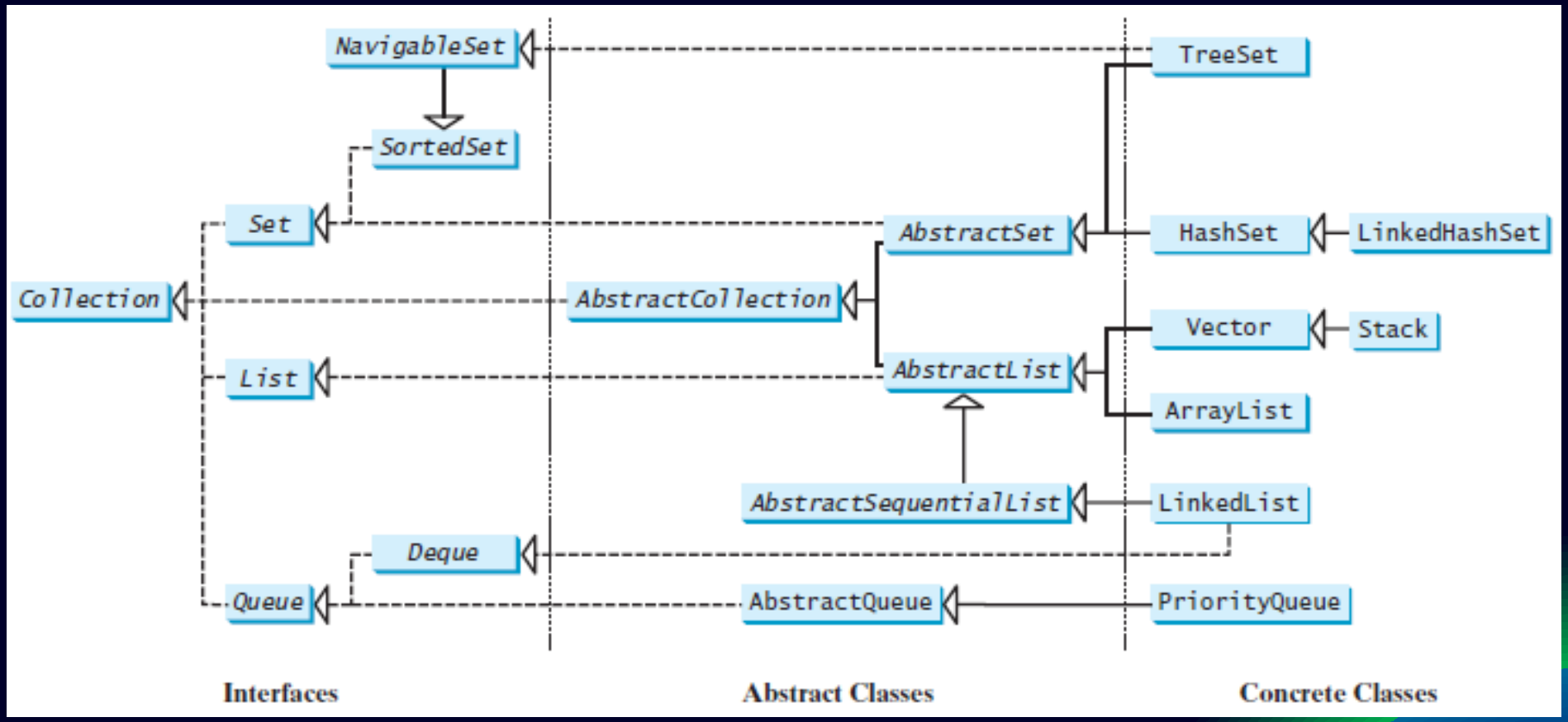
What is a Data Structure?

- 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.
- In object-oriented thinking, a data structure, also known as a container or container object, is an object that stores other objects, referred to as data or elements.

What is a Data Structure?

- The ArrayList class is an example of data structure that stores elements in a list.
- Java provides several more data structures that can be used to organize and manipulate data efficiently.
- These are commonly known as Java Collections Framework.

Java Collection Framework Hierarchy



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

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 two types of containers:
  - One for storing a collection of elements is simply called a collection.
  - The other, for storing key/value pairs, is called a map.

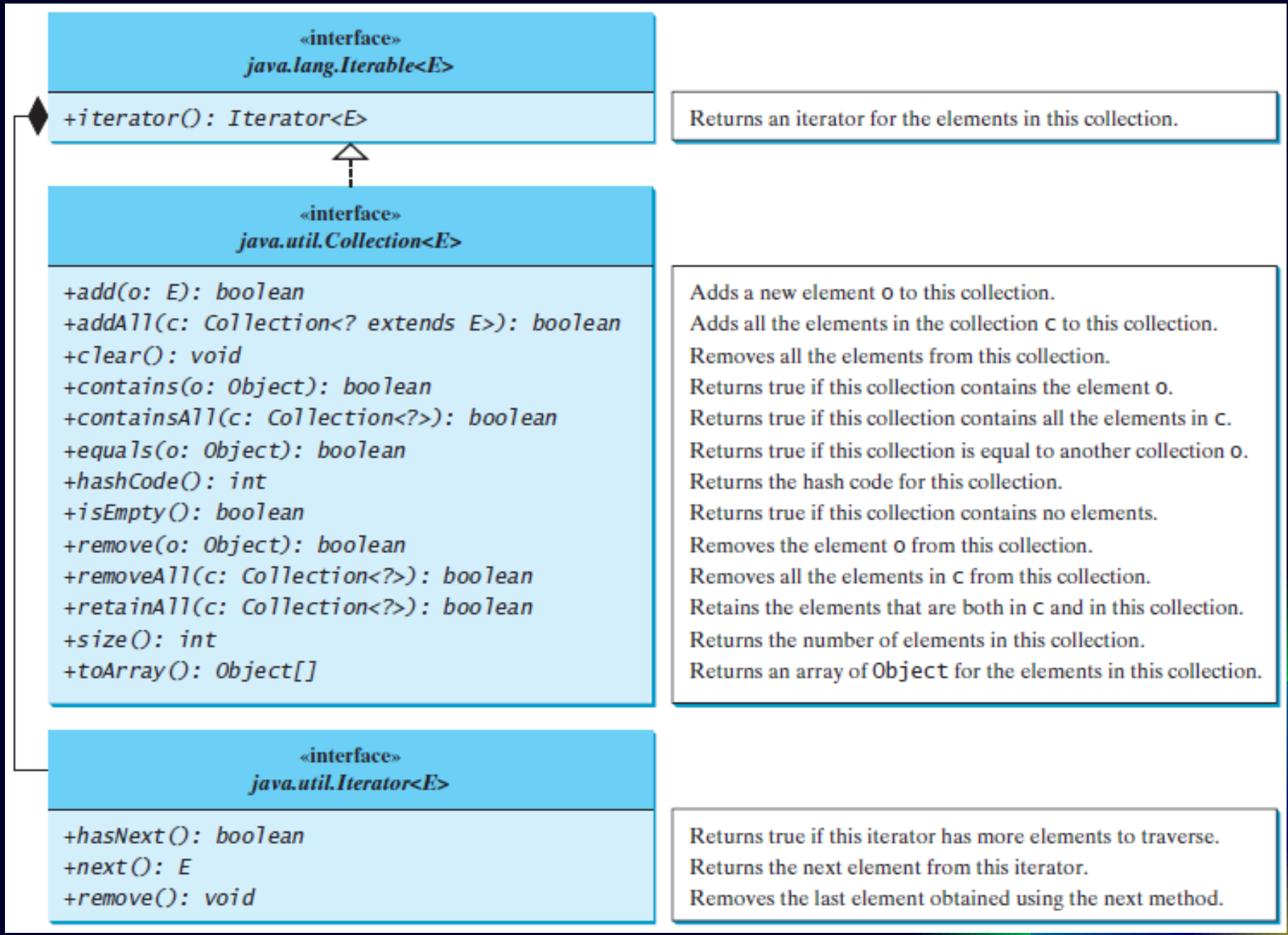
Java Collection Framework hierarchy

There are different kinds of collections.

- Sets store a group of non-duplicate elements.
- Lists store an ordered collection of elements.
- Queues store objects that are processed in first-in, first-out fashion.

The common features of these collections are defined in the interfaces, and implementations are provided in concrete classes,

The Collection Interface



Iterators

- Each collection has an Iterator object that can be used to traverse all the elements in the collection.
- Iterator is a classic design pattern for walking through a data structure without having to expose the details of how data is stored in the data structure and provides a uniform way for traversing elements in various types of collections.

Iterators

- The Collection interface extends the Iterable interface which defines the iterator method, which returns an iterator.
- The iterator method in the Collection interface returns an instance of the Iterator interface which provides sequential access to the elements in the collection using the next() method.
- You can also use the hasNext() method to check whether there are more elements in the iterator, and the remove() method to remove the last element returned by the iterator.

Iterator Example

```
 1  import java.util.*;
 2
 3  public class TestIterator {
 4    public static void main(String[] args) {
 5      Collection<String> collection = new ArrayList<String>();
 6      collection.add("New York");
 7      collection.add("Atlanta");
 8      collection.add("Dallas");
 9      collection.add("Madison");
10
11      Iterator<String> iterator = collection.iterator();
12      while (iterator.hasNext()) {
13        System.out.print(iterator.next().toUpperCase() + " ");
14      }
15      System.out.println();
16    }
17  }
```

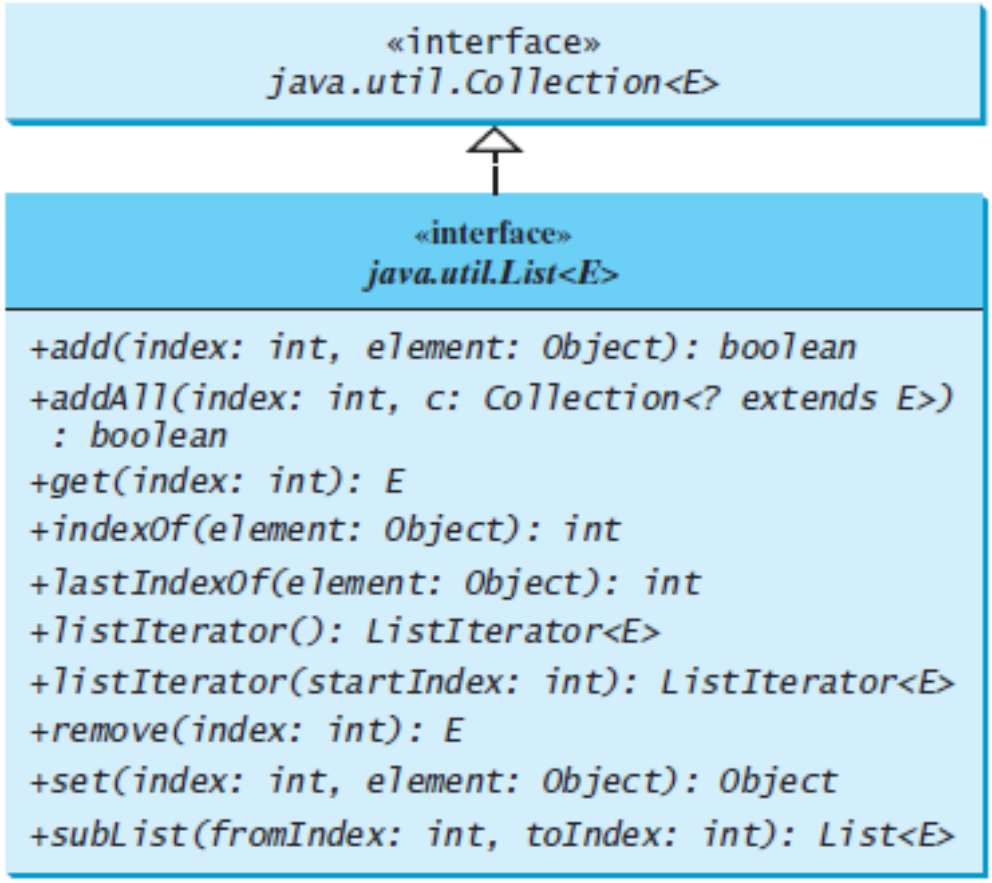
The List Interface

- A set stores non-duplicate elements.
- To allow duplicate elements to be stored in a collection, you need to use a list.
- A list can not only store duplicate elements, but can also allow the user to specify where the element is stored.
- The user can access the element by index.

The List Interface

- The List interface extends the Collection interface and defines a collection for storing elements in a sequential order.
- To create a list, use one of its two concrete classes: ArrayList or LinkedList.

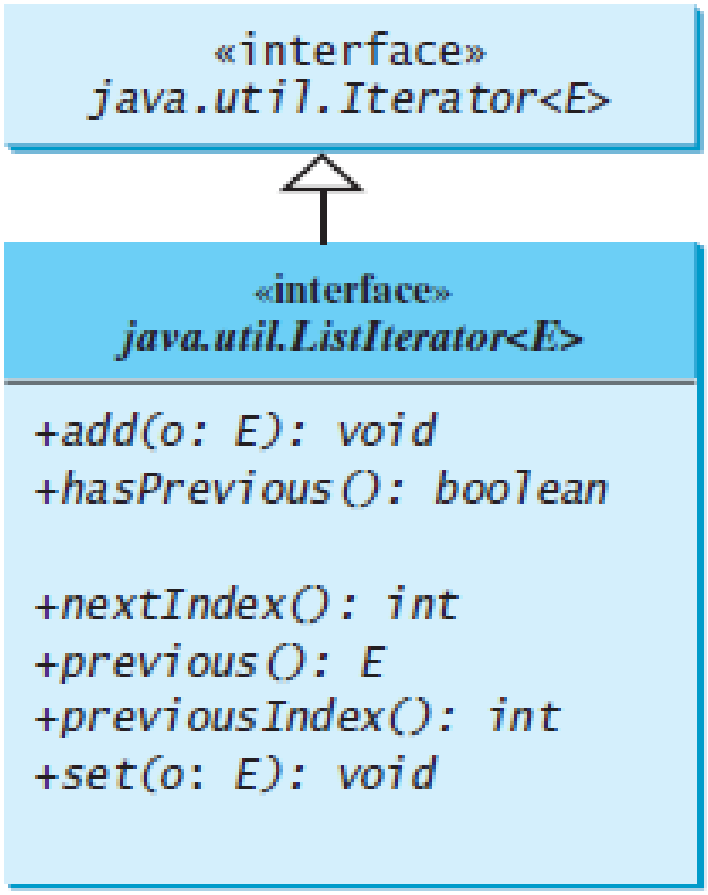
The List Interface



`+add(index: int, element: Object): boolean`	Adds a new element at the specified index.
`+addAll(index: int, c: Collection<? extends E>) : boolean`	Adds all the elements in `c` to this list at the specified index.
`+get(index: int): E`	Returns the element in this list at the specified index.
`+indexOf(element: Object): int`	Returns the index of the first matching element.
`+lastIndexOf(element: Object): int`	Returns the index of the last matching element.
`+listIterator(): ListIterator<E>`	Returns the list iterator for the elements in this list.
`+listIterator(startIndex: int): ListIterator<E>`	Returns the iterator for the elements from `startIndex`.
`+remove(index: int): E`	Removes the element at the specified index.
`+set(index: int, element: Object): Object`	Sets the element at the specified index.
`+subList(fromIndex: int, toIndex: int): List<E>`	Returns a sublist from `fromIndex` to `toIndex-1`.

The List Iterator

The ListIterator interface extends the Iterator interface to add bidirectional traversal of the list.



Adds the specified object to the list.
Returns true if this list iterator has more elements when traversing backward.
Returns the index of the next element.
Returns the previous element in this list iterator.
Returns the index of the previous element.
Replaces the last element returned by the previous or next method with the specified element.

ArrayList and LinkedList

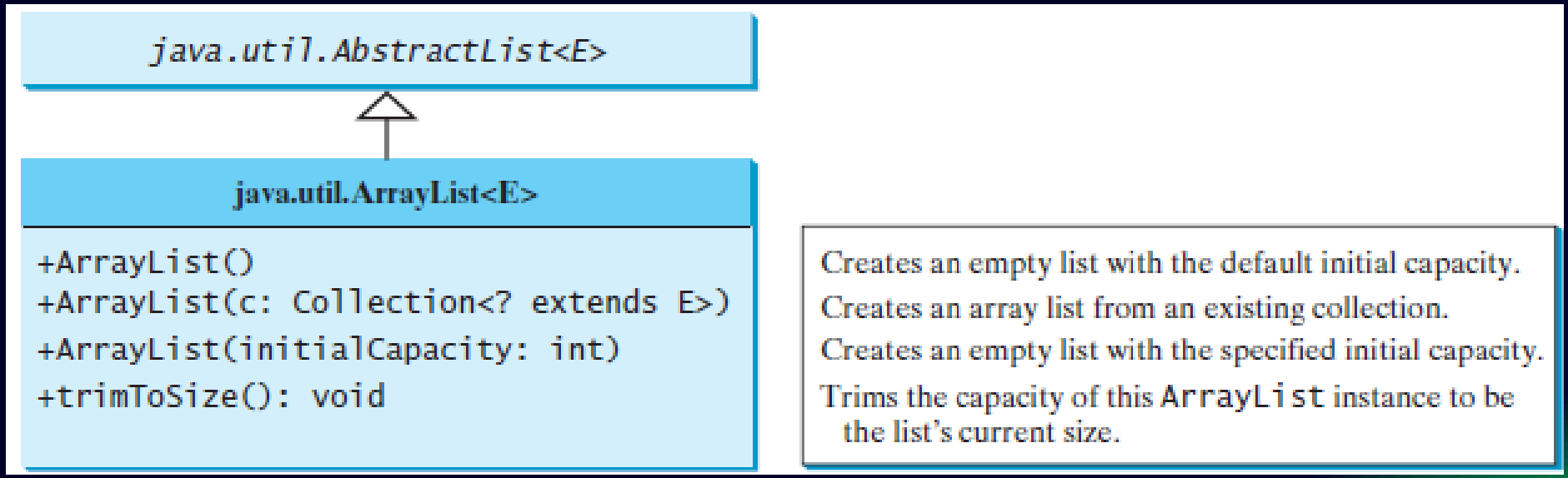
The ArrayList class and the LinkedList class are concrete implementations of the List interface.

- ArrayList stores elements in an array. The array is dynamically created. If the capacity of the array is exceeded, a larger new array is created and all the elements from the current array are copied to the new array.
- LinkedList stores elements in a *linked list*.

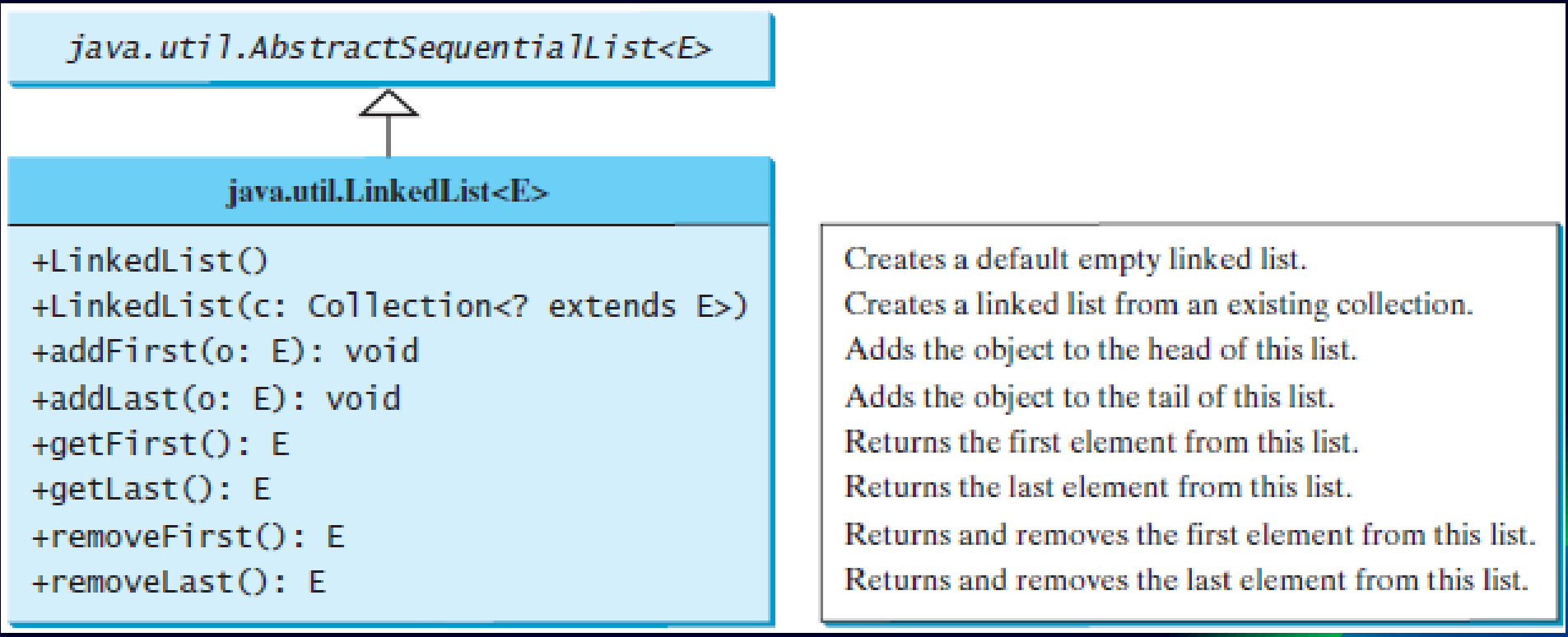
ArrayList and LinkedList

- 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



java.util.LinkedList



Example: Using ArrayList and LinkedList

```
 1  import java.util.*;
 2
 3  public class TestArrayAndLinkedList {
 4    public static void main(String[] args) {
 5      List<Integer> arrayList = new ArrayList<Integer>();
 6      arrayList.add(1); // 1 is autoboxed to new Integer(1)
 7      arrayList.add(2);
 8      arrayList.add(3);
 9      arrayList.add(1);
10      arrayList.add(4);
11      arrayList.add(0, 10);
12      arrayList.add(3, 30);
13
14      System.out.println("A list of integers in the array list:");
15      System.out.println(arrayList);
16
17      LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
18      linkedList.add(1, "red");
19      linkedList.removeLast();
20      linkedList.addFirst("green");
21
22      System.out.println("Display the linked list forward:");
23      ListIterator<Object> listIterator = linkedList.listIterator();
24      while (listIterator.hasNext()) {
25        System.out.print(listIterator.next() + " ");
26      }
27      System.out.println();
28
29      System.out.println("Display the linked list backward:");
30      listIterator = linkedList.listIterator(linkedList.size());
31      while (listIterator.hasPrevious()) {
32        System.out.print(listIterator.previous() + " ");
33      }
34    }
35  }
```

Comparator vs Comparable

- You learned how to compare elements using the Comparable interface.
- Several classes in the Java API, such as String, Date, Calendar, BigInteger, BigDecimal, and all the numeric wrapper classes for the primitive types, implement the Comparable interface.
- The Comparable interface defines the compareTo method, which is used to compare two elements of the same class that implement the Comparable interface.

Comparator vs Comparable

- What if the elements' classes do not implement the **Comparable** interface or the elements have different types? Can these elements be compared?
- Yes, you can define a *comparator* to compare the elements of different classes. To do so, define a class that implements the java.util.Comparator<T> interface.

Comparator vs Comparable

- The Comparator<T> interface has two methods, compare and equals.
  - public int compare(T element1, T 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.
  - public boolean equals(Object element)

    Returns true if the specified object is also a comparator and imposes the same ordering as this comparator.

Comparator Example

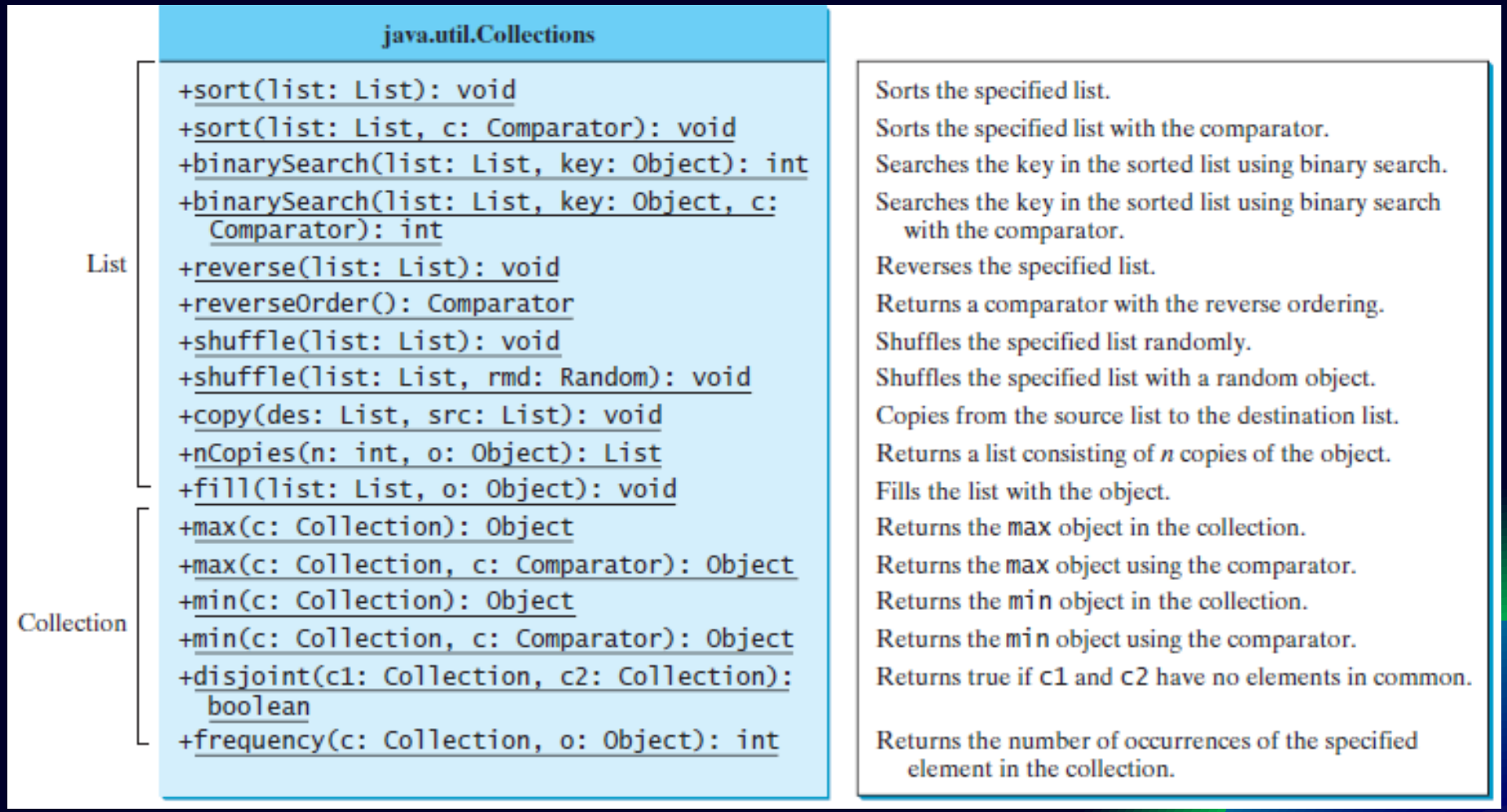
```java
import java.util.Comparator;

public class GeometricObjectComparator
    implements Comparator<GeometricObject> {
  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;
  }
}
```

```java
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 consists exclusively of static methods that operate on or return collections.
- It contains polymorphic algorithms that operate on collections and Lists
- The methods of this class all throw a nullPointerException if the collections or class objects provided to them are null

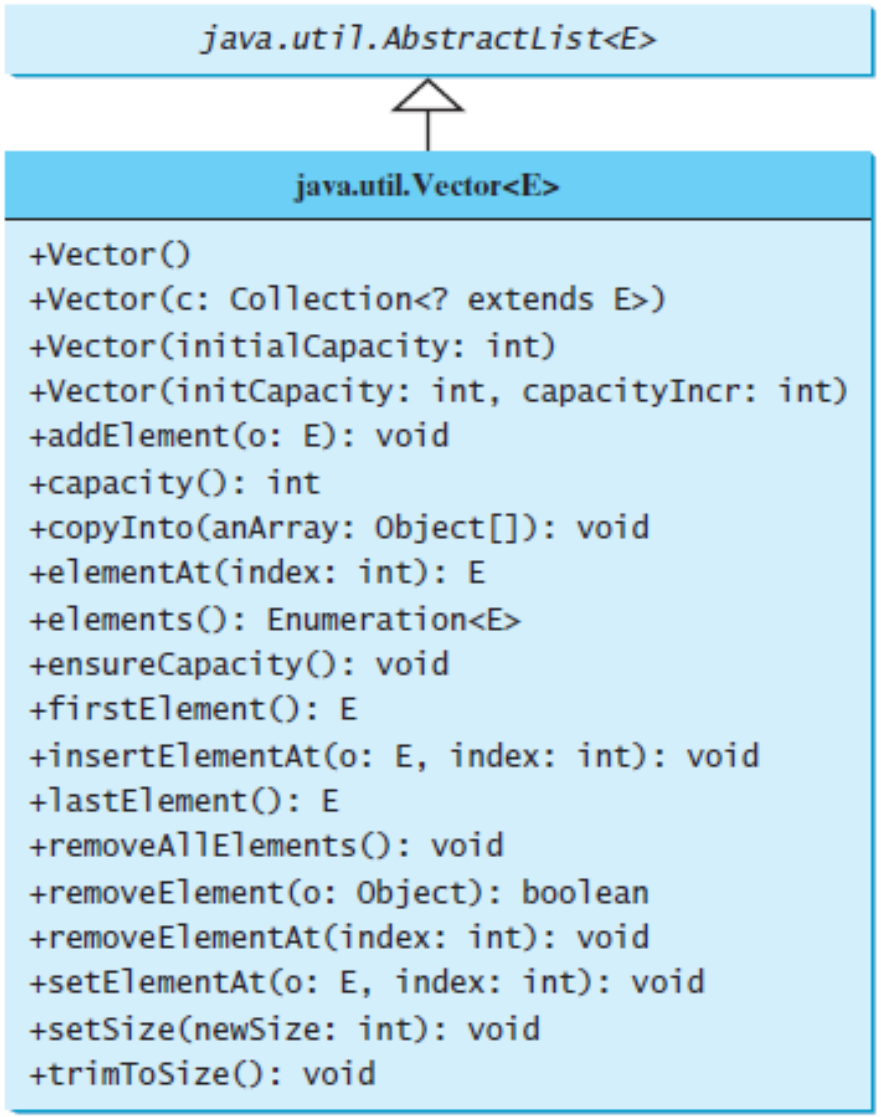
The Collections Class UML Diagram



The Vector and Stack Classes

- Vector is a subclass of AbstractList, and Stack is a subclass of Vector in the Java API.
- Vector is the same as ArrayList, except that Vector contains the synchronized methods for accessing and modifying the vector.
- None of the collection data structures introduced so far are synchronized.
- If synchronization is required, you can use the synchronized versions of the collection classes (Chapter 30).

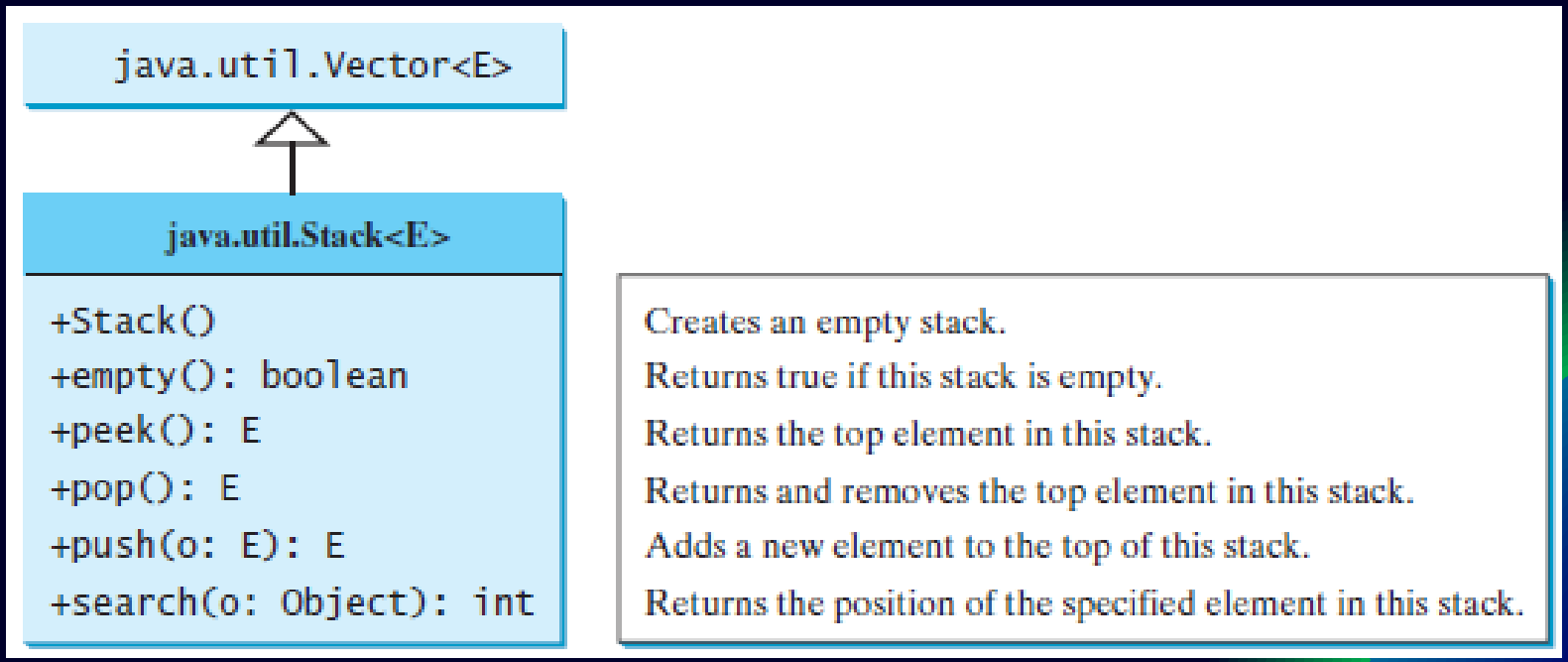
The Vector Class UML Diagram



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

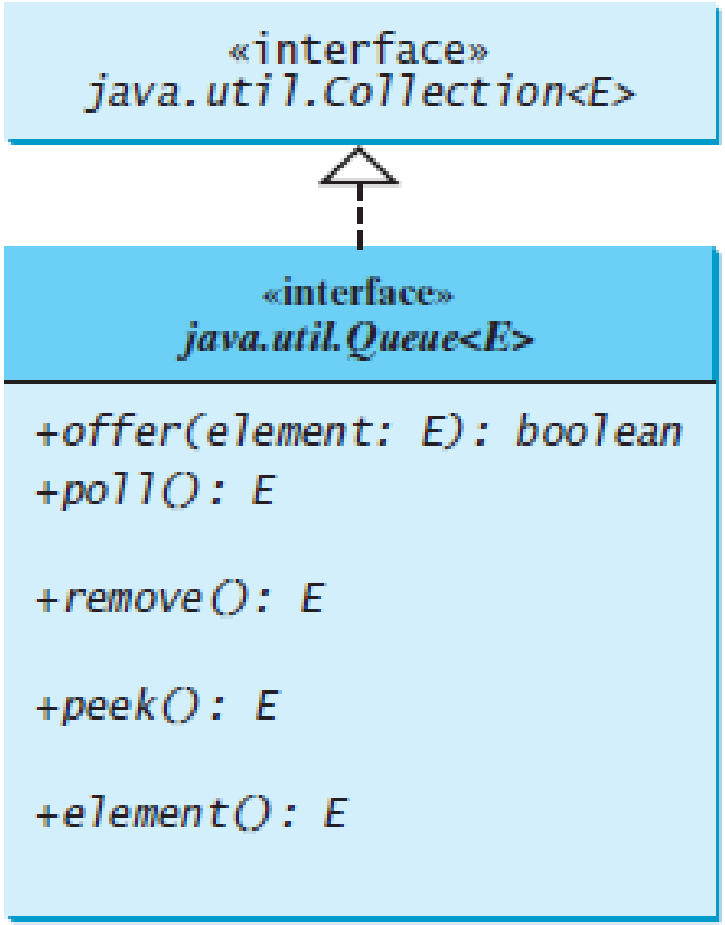
- The Stack class represents a last-in-first-out stack of objects.
- The elements are accessed only from the top of the stack.
- You can retrieve, insert, or remove an element from the top of the stack.



Queues and Priority Queues

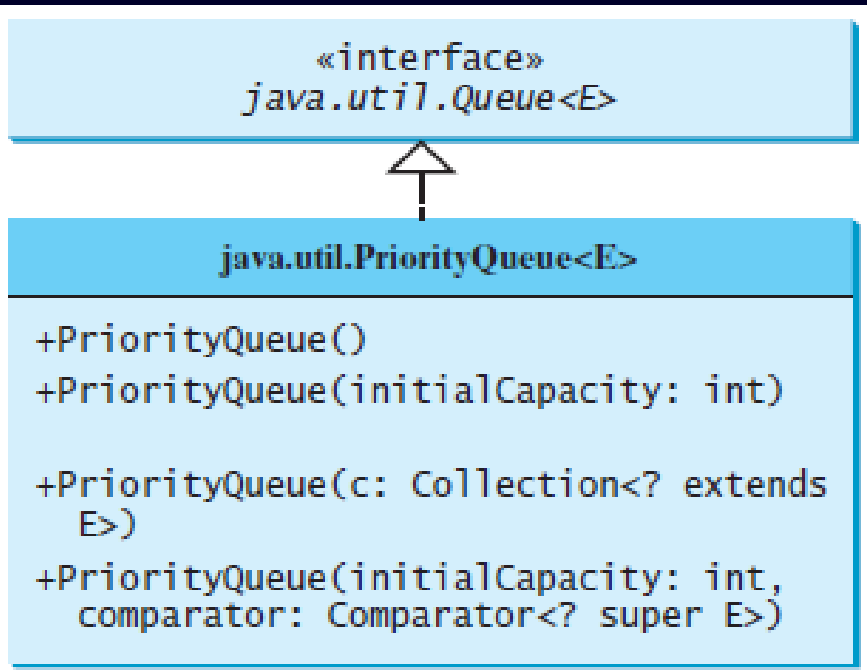
- A queue is a first-in/first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue.
- In a priority queue, elements are assigned priorities based on their natural ordering or by a Comparator provided at queue construction time. When accessing 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, throwing an exception if this queue is empty.

The PriorityQueue Class



Creates a default priority queue with initial capacity 11.

Creates a default priority queue with the specified initial capacity.

Creates a priority queue with the specified collection.

Creates a priority queue with the specified initial capacity and the comparator.

Programming Challenge

(Use iterators on linked lists) Write a test program that stores 5 million integers in a linked list and test the time to traverse the list using an iterator vs. using the get(index) method with a loop.

Programming Challenge

(Match grouping symbols) A Java program contains various pairs of grouping symbols, such as:

- Parentheses: (and)
- Braces: { and }
- Brackets: [and]

Note that the grouping symbols cannot overlap.

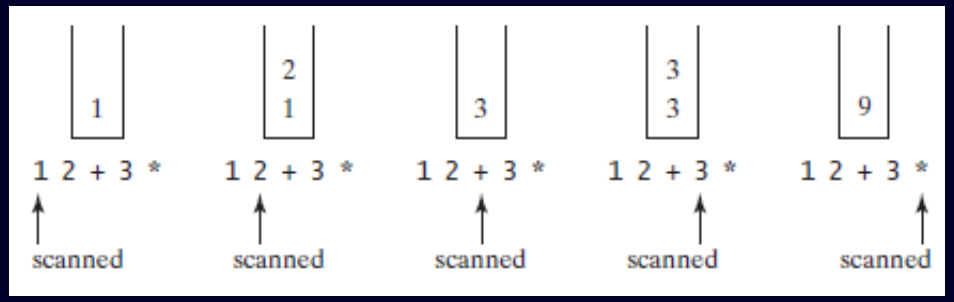
For example, (a{b)} is illegal.

Write a program to check whether a Java source-code file has correct pairs of grouping symbols.

Hint: Stack

Programming Challenge

(Postfix notation) Postfix notation is a way of writing expressions without using parentheses. For example, the expression (1 + 2) * 3 would be written as 1 2 + 3 *. A postfix expression is evaluated using a stack. Scan a postfix expression from left to right. A variable or constant is pushed into the stack. When an operator is encountered, apply the operator with the top two operands in the stack and replace the two operands with the result. The following diagram shows how to evaluate 1 2 + 3 *.



Write a program to evaluate postfix expressions.