Lists, Stacks, Queues, and Priority Queues

Paul Fodor
CSE260, Computer Science B: Honors
Stony Brook University
http://www.cs.stonybrook.edu/~cse260

Objectives

- To explore the relationship between interfaces and classes in the **Java Collections Framework** hierarchy.
- To use the common methods defined in the **`Collection`** interface for operating collections.
- To use the **`Iterator`** interface to traverse the elements in a collection.
- To use a **`for`**-each loop to traverse the elements in a collection.
- To explore how and when to use **`ArrayList`** or **`LinkedList`** to store elements.
- To compare elements using the **`Comparable`** interface and the **`Comparator`** interface.
- To use the static utility methods in the **`Collections`** class for sorting, searching, shuffling lists, and finding the largest and smallest element in collections.
- To distinguish between **`Vector`** and **`ArrayList`** and to use the **`Stack`** class for creating stacks.
- To explore the relationships among **`Collection`**, **`Queue`**, **`LinkedList`**, and **`PriorityQueue`** and to create priority queues using the **`PriorityQueue`** class.
- To use stacks to write a program to evaluate expressions.

Data structures

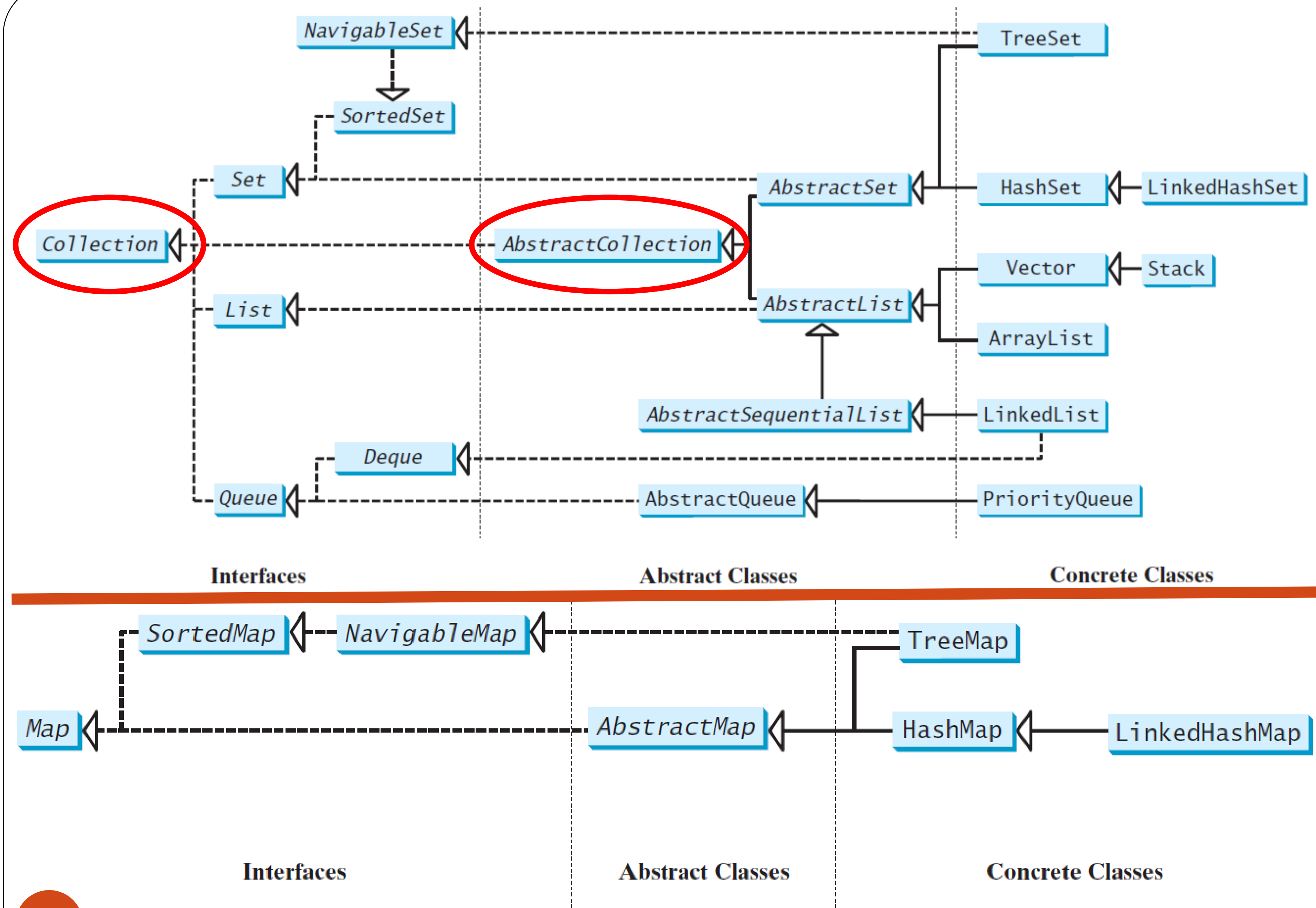
- A ***data structure*** or ***collection*** is a collection of data organized in some fashion
  - not only stores data but also supports operations for accessing and manipulating the data
    - Choosing the best data structures and algorithms for a particular task is one of the keys to developing high-performance software
- In object-oriented thinking, a data structure, also known as a ***container***, is an object that stores other objects, referred to as ***elements***

Java Collection Framework hierarchy

- Java provides several data structures that can be used to organize and manipulate data efficiently, commonly known as ***Java Collections Framework***
- The Java Collections Framework supports two types of containers:
  - One for ***storing*** a collection of elements, simply called a ***collection***
    - ***Lists*** store an ordered collection of elements
    - ***Sets*** store a group of nonduplicate elements
    - ***Stacks*** store objects that are processed in a last-in, first-out fashion
    - ***Queues*** store objects that are processed in a first-in, first-out fashion
    - ***PriorityQueues*** store objects that are processed in the order of their priorities
  - One for storing key/value pairs, called a ***map***
    - Note: this is called a ***dictionary*** in Python

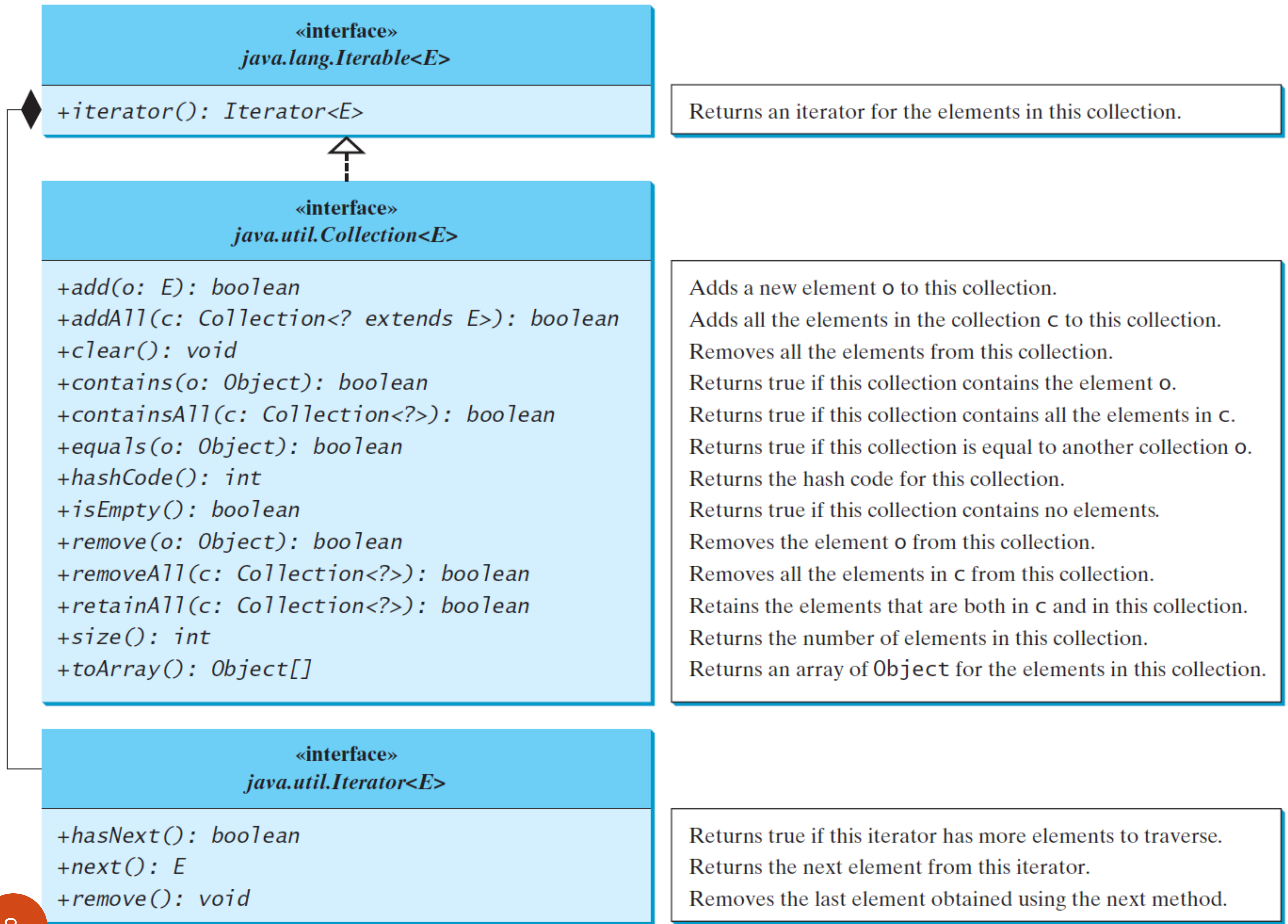
Java Collection Framework hierarchy

- All the interfaces and classes defined in the Java Collections Framework are grouped in the **`java.util`** package
- The design of the Java Collections Framework is an excellent example of using interfaces, abstract classes, and concrete classes
  - The interfaces define the framework/general API
  - The abstract classes provide partial implementation
    - Providing an abstract class that partially implements an interface makes it convenient for the user to write the code for the specialized containers
      - **`AbstractCollection`** is provided for convenience (for this reason, it is called ***a convenience abstract class***)
  - The concrete classes implement the interfaces with concrete data structures


NavigableSet
SortedSet
Set
Collection
List
Deque
Queue
AbstractCollection
AbstractSet
AbstractList
AbstractSequentialList
AbstractQueue
TreeSet
HashSet
LinkedHashSet
Vector
Stack
ArrayList
LinkedList
PriorityQueue
Interfaces
Abstract Classes
Concrete Classes
SortedMap
NavigableMap
Map
AbstractMap
TreeMap
HashMap
LinkedHashMap
Interfaces
Abstract Classes
Concrete Classes

Java Collection Framework hierarchy

- The **`Collection`** interface is the root interface for manipulating a collection of objects
  - The **`AbstractCollection`** class provides partial implementation for the **`Collection`** interface (all the methods in **`Collection`** except the **`add`**, **`size`**, and **`iterator`** methods)
- Note: the **`Collection`** interface implements the **`Iterable`** interface
  - We can obtain an **`Iterator`** object for traversing elements in the collection
    - Also used by for-each loops


«interface»
java.lang.Iterable<E>
+iterator(): Iterator<E>
Returns an iterator for the elements in this collection.
«interface»
java.util.Collection<E>
+add(o: E): boolean
+addAll(c: Collection<? extends E>): boolean
+clear(): void
+contains(o: Object): boolean
+containsAll(c: Collection<?>): boolean
+equals(o: Object): boolean
+hashCode(): int
+isEmpty(): boolean
+remove(o: Object): boolean
+removeAll(c: Collection<?>): boolean
+retainAll(c: Collection<?>): boolean
+size(): int
+toArray(): Object[]
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.
«interface»
java.util.Iterator<E>
+hasNext(): boolean
+next(): E
+remove(): void
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.

- **Example of using the methods in the Java Collection Framework:**

```
import java.util.*;
public class TestCollection {
  public static void main(String[] args) {
    ArrayList<String> collection1 = new ArrayList<>();
    collection1.add("New York"); // add
    collection1.add("Atlanta");
    collection1.add("Dallas");
    collection1.add("Madison");

    System.out.println("A list of cities in collection1:");
    System.out.println(collection1);

    // the Collection interface's contains method
    System.out.println("\nIs Dallas in collection1? "
      + collection1.contains("Dallas")); // contains

    // the Collection interface's remove method
    collection1.remove("Dallas"); // remove

    // the Collection interface's size method
    System.out.println("\n" + collection1.size() + // size
       " cities are in collection1 now");
```

```
Collection<String> collection2 = new ArrayList<>();
collection2.add("Seattle");
collection2.add("Portland");

System.out.println("\nA list of cities in collection2:");
System.out.println(collection2);

ArrayList<String> c1 = (ArrayList<String>)
          (collection1.clone()); // clone
c1.addAll(collection2); // addAll
System.out.println("\nCities in collection1 or collection2:");
System.out.println(c1);

c1 = (ArrayList<String>)(collection1.clone());
c1.retainAll(collection2); // retainAll
System.out.print("\nCities in collection1 and collection2:");
System.out.println(c1);

c1 = (ArrayList<String>)(collection1.clone());
c1.removeAll(collection2); // removeAll
System.out.print("\nCities in collection1, but not in 2: ");
System.out.println(c1);
  }
}
```

```
Output:

A list of cities in collection1:
[New York, Atlanta, Dallas, Madison]

Is Dallas in collection1? true

3 cities are in collection1 now

A list of cities in collection2:
[Seattle, Portland]

Cities in collection1 or collection2:
[New York, Atlanta, Madison, Seattle, Portland]

Cities in collection1 and collection2:[]

Cities in collection1, but not in 2: [New York, Atlanta,
Madison]
```

Java Collection Framework hierarchy

- All the concrete classes in the Java Collections Framework implement the **`java.lang.Cloneable`** and **`java.io.Serializable`** interfaces except that **`java.util.PriorityQueue`** does not implement the **`Cloneable`** interface
- Some of the methods in the **`Collection`** interface cannot be implemented in the concrete subclass (e.g., the read-only collections cannot add or remove)
  - In this case, the method would throw **`java.lang.UnsupportedOperationException`**, like this:

```
public void someMethod() {
        throw new UnsupportedOperationException
                ("Method not supported");
}
```

Iterators

- Each collection is **`Iterable`**
    - ***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
        - Also used in for-each loops:

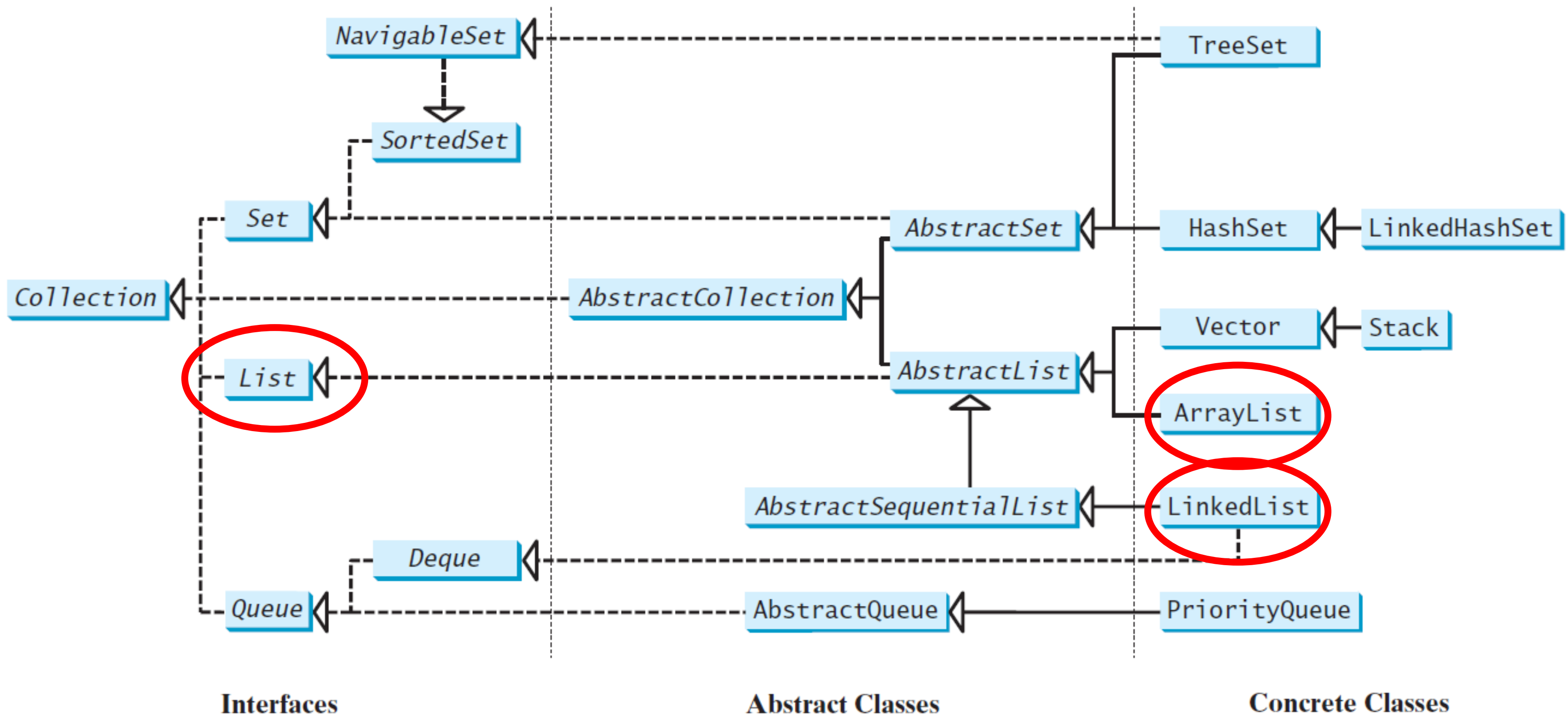
```
for(String element: collection)
      System.out.print(element + " ");
```

- The **`Collection`** interface extends the **`Iterable`** interface
    - You can obtain a collection **`Iterator`** object to traverse all the elements in the collection with the **`iterator()`** method in the **`Iterable`** interface which returns an instance of **`Iterator`**
        - The **`Iterable`** interface defines the **`iterator`** method, which returns an **`Iterator`**

```java
import java.util.*;

public class TestIterator {
  public static void main(String[] args) {
    Collection<String> collection = new ArrayList<>();
    collection.add("New York");
    collection.add("Atlanta");
    collection.add("Dallas");
    collection.add("Madison");

    Iterator<String> iterator = collection.iterator();
    while (iterator.hasNext()) {
      System.out.print(iterator.next().toUpperCase() + " ");
    }

    System.out.println();
  }
}
```

Output: **NEW YORK ATLANTA DALLAS MADISON**

The **`List`** Interface



The `List` Interface

- A *list* collection stores elements in a **sequential** order, and allows the user to specify where the element is stored
- The user can also access the elements by ***index***
- The **`List`** interface stores elements in sequence and permits duplicates
- Two concrete classes in Java Collections Framework: **`ArrayList`** and **`LinkedList`**

The `List` Interface

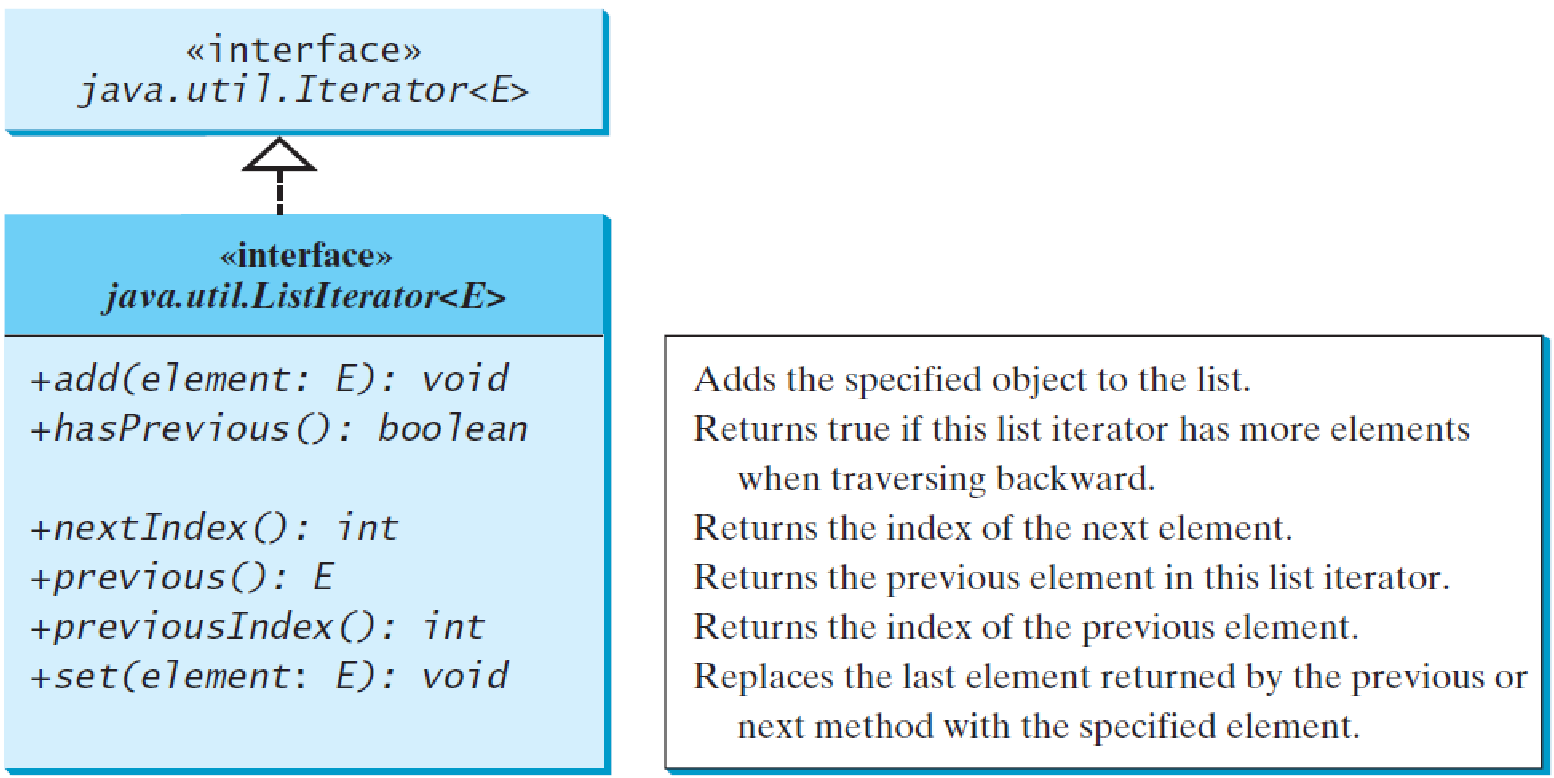
«interface»
java.util.Collection<E>

«interface»
java.util.List<E>

+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 `ListIterator`

- The **`listIterator()`** and **`listIterator(startIndex)`** methods return an instance of **`ListIterator`**
  - The **`ListIterator`** interface extends the **`Iterator`** interface for bidirectional traversal of the list and add elements to the list



The `ListIterator`

- The **`nextIndex()`** method returns the index of the next element in the iterator, and the **`previousIndex()`** returns the index of the previous element in the iterator
- The **`add(element)`** method inserts the specified element into the list immediately before the next element that would be returned by the **`next()`** method defined in the **`Iterator`** interface

ArrayList and **LinkedList**

- The **ArrayList** class and the **LinkedList** class are concrete implementations of the **List** interface
  - A list can grow or shrink dynamically
    - While 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 an **array**

java.util.ArrayList

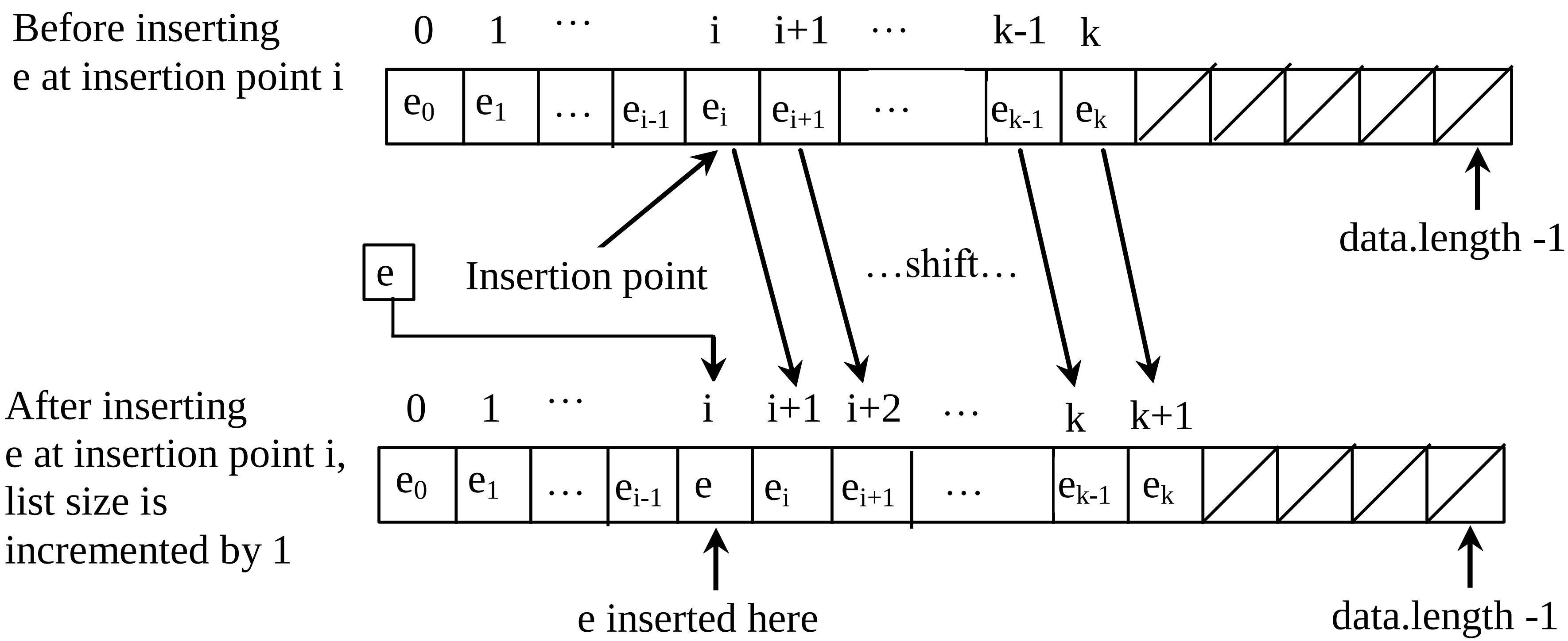
«interface»
java.util.Collection<E>

«interface»
java.util.List<E>

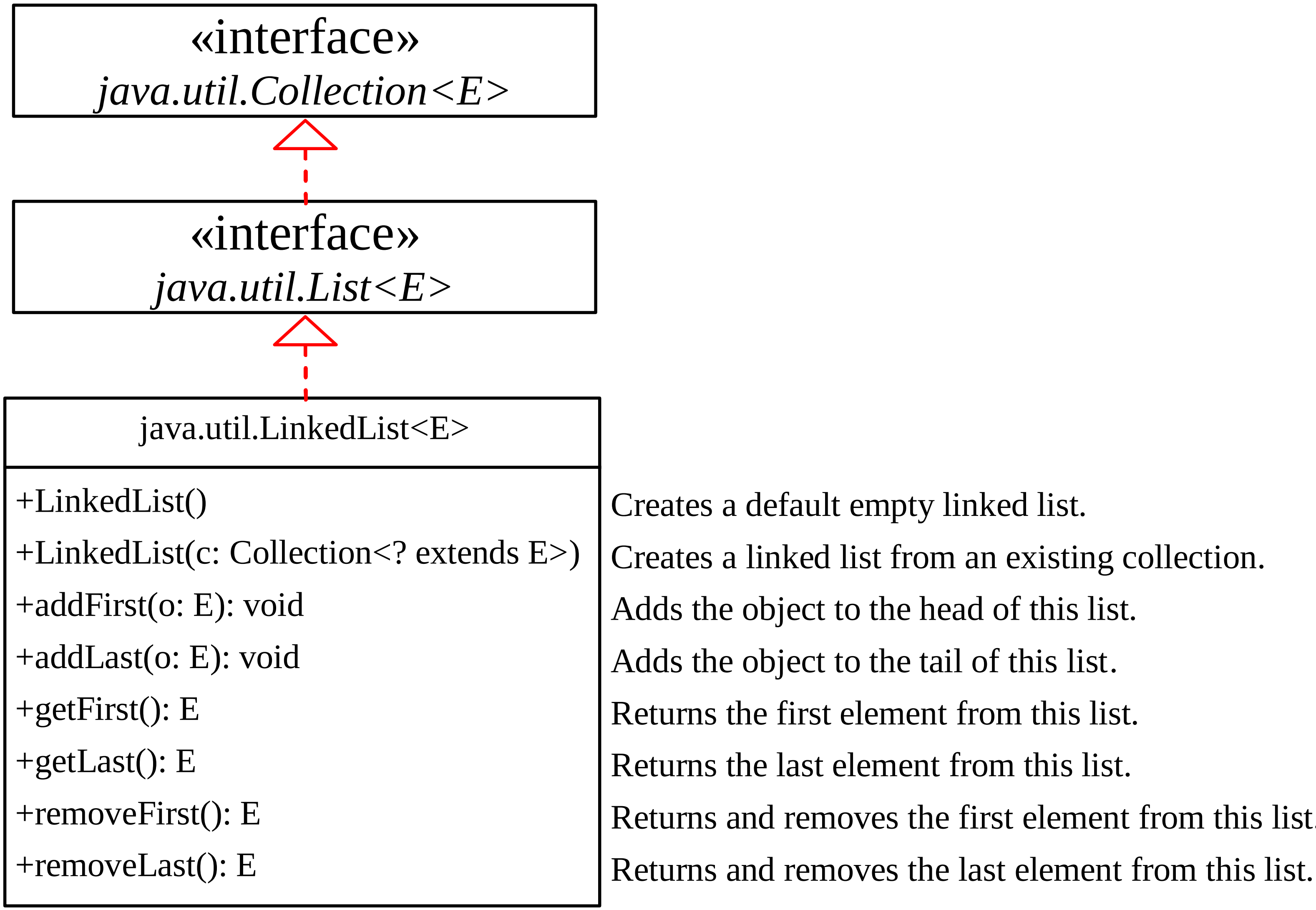
java.util.ArrayList<E>	
+ArrayList()	Creates an empty list with the default initial capacity.
+ArrayList(c: Collection<? extends E>)	Creates an array list from an existing collection.
+ArrayList(initialCapacity: int)	Creates an empty list with the specified initial capacity.
+trimToSize(): void	Trims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

java.util.ArrayList

- Implemented with arrays, e.g., before inserting a new element at a specified index, shift all the elements after the index to the right and increase the ArrayList size by 1

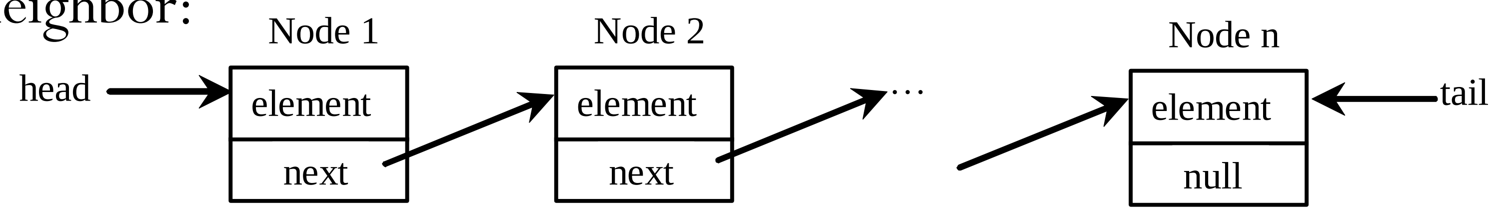


java.util.LinkedList



Nodes in Linked Lists

- A linked list **consists of nodes:**
  - Each node contains an element/value, and each node is ***linked*** to its next neighbor:



  - A node can be defined as a class, as follows:

```
class Node<E> {
  E element;
  Node<E> next;
  public Node(E o) {
    element = o;
  }
}
```

ArrayList and LinkedList

- Which of the two classes **ArrayList** class and the **LinkedList** class you use depends on your specific needs:
  - The critical difference between them pertains to internal implementation, which affects their performance.
    - 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 your application requires the insertion or deletion of elements from **at the beginning of the list**, you should choose **LinkedList**

Using **`ArrayList`** and **`LinkedList`**

- The next example creates an **`ArrayList`** filled with numbers, and inserts new elements into the specified location in the list
- The example also creates a **`LinkedList`** from the array list, inserts and removes the elements from the list
- Finally, the example traverses the list forward and backward
- Note: A list can hold identical elements: Integer 1 is stored twice in the list.

```java
import java.util.*;

public class TestArrayAndLinkedList {
  public static void main(String[] args) {
    List<Integer> arrayList = new ArrayList<>();
    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<>(arrayList);
    linkedList.add(1, "red");
    linkedList.removeLast();
    linkedList.addFirst("green");
```

```java
    System.out.println("Display the linked list backward with index:");
    for (int i = linkedList.size() - 1; i >= 0; i--) {
      System.out.print(linkedList.get(i) + " ");
    }
    System.out.println();

    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() + " ");
    }
  }
}
```

- Output:

```
A list of integers in the array list:
[10, 1, 2, 30, 3, 1, 4]

Display the linked list backward with index:
1 3 30 2 1 red 10 green

Display the linked list forward:
green 10 red 1 2 30 3 1

Display the linked list backward:
1 3 30 2 1 red 10 green
```

`ArrayList` and `LinkedList`

- The **`get(i)`** method is available for a **linked list**, but it is a more time-consuming operation to find each element
  - Instead you should use an **iterator** or **for-each** loops:

```
for (int i = 0; i < list.size(); i++) {
  process list.get(i);
}
```

(a) Very inefficient

```
for (listElementType s: list) {
  process s;
}
```

(b) Efficient

The **`Comparator`** Interface

- Sometimes you want to compare the elements that are not instances of **`Comparable`** or by a <u>different criteria</u> than **`Comparable`**
- You can define a **`Comparator`** to compare these elements
  - Define a class that implements the **`java.util.Comparator<T>`** interface
  - The **`Comparator`** 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

```java
import java.util.Comparator;

public class GeometricObjectComparator
        implements Comparator<GeometricObject>,
                   java.io.Serializable {
// It is generally a good idea for comparators to implement
//  Serializable, as they may be used as ordering methods in
//  serializable data structures.
    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;
  }
```

Static Methods for Lists and Collections

- The **java.util.Collections** class contains static methods to perform common operations in a collection or a list
  - **max**, **min**, **disjoint**, and **frequency** methods for collections
  - **sort**, **binarySearch**, **reverse**, **shuffle**, **copy**, and **fill** methods for lists

```
static <T extends Comparable<? super T>> void
        sort(List<T> list)
```

uses the **compareTo** method in the **Comparable** interface

```
static <T extends Comparator<? super T>> void
        sort(List<T> list, Comparator<T> c)
```

uses the **compare** method in the **Comparator** interface

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.

Other **Collections** class useful static methods:

- **rotate(List list, int distance)** - Rotates all of the elements in the list by the specified distance.
- **replaceAll(List list, Object oldVal, Object newVal)** - Replaces all occurrences of one specified value with another.
- **indexOfSubList(List source, List target)** - Returns the index of the first sublist of source that is equal to target.
- **lastIndexOfSubList(List source, List target)** - Returns the index of the last sublist of source that is equal to target.
- **swap(List, int, int)** - Swaps the elements at the specified positions in the specified list.
- **addAll(Collection<? super T>, T...)** - Adds all of the elements in the specified array to the specified collection.

Static Methods for Lists

- **Sorting**:

```
List<String> list = Arrays.asList("red", "green",
                                  "blue");
Collections.sort(list);
System.out.println(list);
```

The output is: `[blue, green, red]`

- To sort it in descending order, you can simply use the `Collections.reverseOrder()` method to return a `Comparator` object that orders the elements in reverse of their natural order

```
Collections.sort(list,
     Collections.reverseOrder());
System.out.println(list);
```

The output is `[yellow, red, green, blue]`

Static Methods for Lists

- **Binary search:**
  - You can use the **`binarySearch`** method to search for a key in a sorted list
    - To use this method, the list must be sorted in **increasing** order
    - If the **`key`** is not in the list, the method returns **`-(insertion point + 1)`**

```
List<Integer> list1 =
   Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
System.out.println("(1) Index: " +
   Collections.binarySearch(list1, 7));  //2
System.out.println("(2) Index: " +
   Collections.binarySearch(list1, 9));  //-4
List<String> list2 = Arrays.asList("blue", "green", "red");
System.out.println("(3) Index: " +
   Collections.binarySearch(list2, "red")); //2
System.out.println("(4) Index: " +
   Collections.binarySearch(list2, "cyan")); //-2
```

Static Methods for Lists and Collections

- **Reverse:**

```
List<String> list =
     Arrays.asList("yellow", "red",
          "green", "blue");
Collections.reverse(list);
System.out.println(list);
```

The code displays: `[blue, green, red, yellow]`

Static Methods for Lists and Collections

- **Shuffle:**

```
List<String> list =
  Arrays.asList("yellow", "red", "green", "blue");
Collections.shuffle(list);
System.out.println(list);
```

- You can also use the `shuffle(List, Random)` method to randomly reorder the elements in a list with a specified `Random` object.
  - Using a specified `Random` object is useful to shuffle another list with an identical sequences of elements

Static Methods for Lists and Collections

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Random;
public class SameShuffle {
   public static void main(String[] args) {
         List<String> list1 = Arrays.asList("yellow", "red", "green",
               "blue");
         List<String> list2 = Arrays.asList("Y", "R", "G", "B");
         Collections.shuffle(list1, new Random(20));
         Collections.shuffle(list2, new Random(20));
         System.out.println(list1);
         System.out.println(list2);
   }
}
```

```
[blue, yellow, red, green]
[B, Y, R, G]
```

Static Methods for Lists and Collections

- **`copy(dest, src)`**: <u>copy</u> all the elements from a source list to a destination list on the same index

```
List<String> list1 = Arrays.asList("yellow",
        "red", "green", "blue");
List<String> list2 = Arrays.asList("white",
        "black");
Collections.copy(list1, list2);
System.out.println(list1);
```

The output for list1 is **`[white, black, green, blue]`**.

  - The **`copy`** method performs a **<u>shallow</u>** copy: only the references of the elements from the source list are copied

Static Methods for Lists and Collections

- If the destination list is smaller than the source list then we get a runtime error:

```
List<String> list2 = Arrays.asList("yellow",
        "red", "green", "blue");
List<String> list1 = Arrays.asList("white",
        "black");
Collections.copy(list1, list2);
```

Runtime error:

```
java.lang.IndexOutOfBoundsException: Source
does not fit in destination
```

Arrays$ArrayList

- Java provides the static **asList** method for creating a list from a variable-length list of arguments

```
List<String> list1 = Arrays.asList("red",
  "green", "blue");
List<Integer> list2 = Arrays.asList(10, 20,
  30, 40, 50);
```

returns a **List** reference of inner class object defined within **Arrays** :

java.util.Arrays$ArrayList, which is also called **ArrayList** but it is just a wrapper for an array

Static Methods for Lists and Collections

- You can use the **`nCopies(int n, Object o)`** method to create an immutable list that consists of **`n`** copies of the specified object

```
List<GregorianCalendar> list1 =
  Collections.nCopies(3,
    new GregorianCalendar(2022,0,1));
```

  - **`list1`** is a list with three **`Calendar`** objects.
  - The list created from the **`nCopies`** method is immutable, so you cannot **`add/remove`** elements in the list --- All the elements have the same reference!

Static Methods for Lists and Collections

```
import java.util.*;
public class Test {
       public static void main(String[] args) {
              List<GregorianCalendar> list1 =
                             Collections.nCopies(3,
                               new GregorianCalendar(2022,0,1));
              list1.get(0).set(Calendar.YEAR, 2024);
              for(GregorianCalendar g:list1)
                     System.out.println(g.get(Calendar.YEAR));
       }
}
2024
2024
2024
```

Static Methods for Lists and Collections

```
List<GregorianCalendar> list1 =
  Collections.nCopies(3,
     new GregorianCalendar(2020,0,1));
```

- **`list1`** is an instance of an inner class of **`Collections`**: class **`java.util.Collections$CopiesList`**

Static Methods for Lists and Collections

- `fill(List list, Object o)` method replaces all the elements in the list with the specified element

```
List<String> list =
  Arrays.asList("red","green","blue");
Collections.fill(list, "black");
System.out.println(list);
```

  - Output: `[black, black, black]`

Static Methods for Lists and Collections

- The **`max`** and **`min`** methods find the maximum and minimum elements in a collection

```
Collection<String> collection =
        Arrays.asList("red", "green", "blue");
System.out.println(Collections.max(collection));
System.out.println(Collections.min(collection));
```

Static Methods for Lists and Collections

- **`disjoint(collection1, collection2)`** method returns **`true`** if the two collections have no elements in common

```
Collection<String> collection1 = Arrays.asList("red", "cyan");
Collection<String> collection2 = Arrays.asList("red", "blue");
Collection<String> collection3 = Arrays.asList("pink", "tan");
System.out.println(Collections.disjoint(collection1,
      collection2)); // false
System.out.println(Collections.disjoint(collection1,
      collection3)); // true
```

Static Methods for Lists and Collections

- **`frequency(collection, element)`** method finds the number of occurrences of the **`element`** in the **`collection`**

```
Collection<String> collection =
    Arrays.asList("red", "cyan", "red");
System.out.println(
      Collections.frequency(collection,"red"));
```

returns **2**

Static Methods for Lists and Collections

```
Collection<String> collection =
    Arrays.asList(new String("red"),
     "cyan", new String("red"), "red");
System.out.println(
 Collections.frequency(collection,"red"));
```

returns 3 because it uses the `.equals` method

Static Methods for Lists and Collections

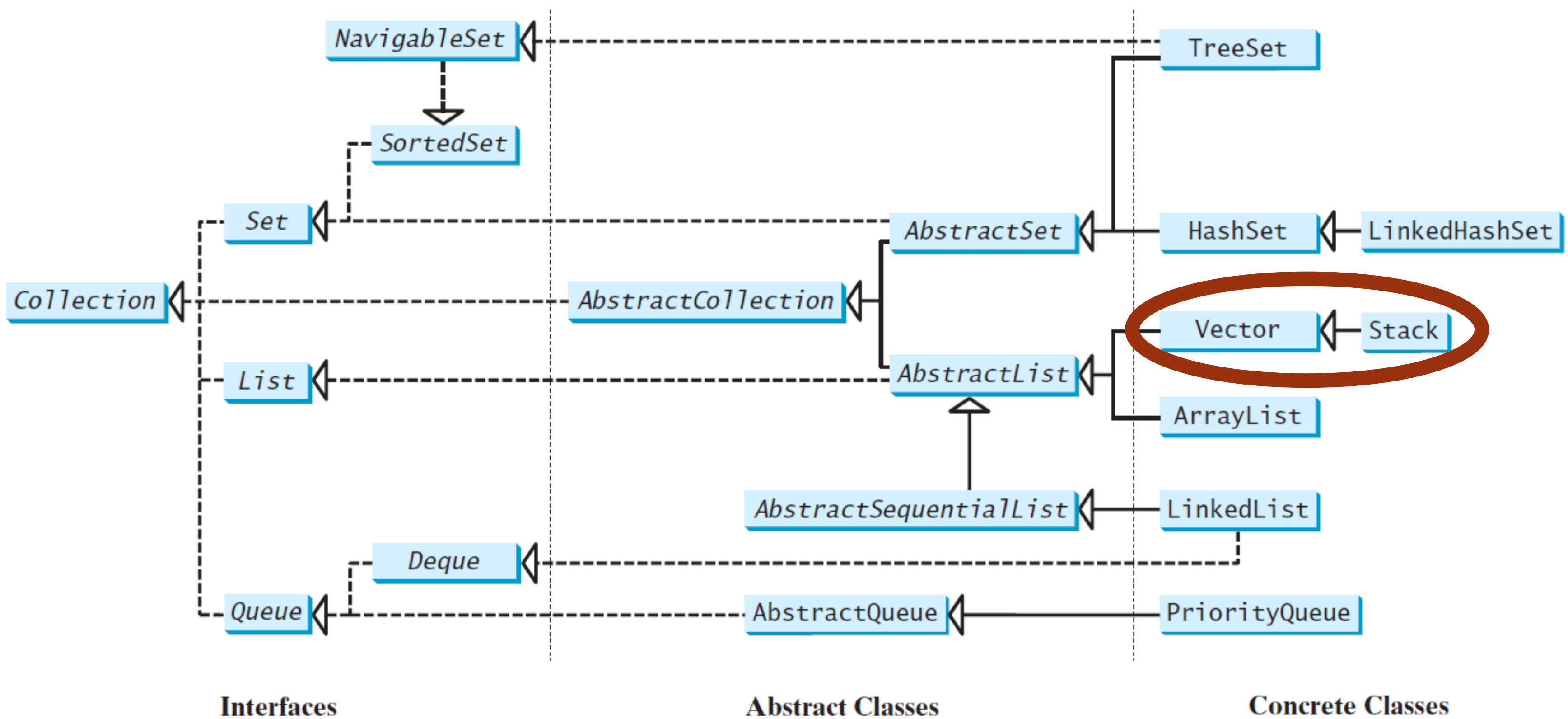
Also

```
Collection<String> collection =
    Arrays.asList("red", "cyan", "red");
System.out.println(
 Collections.frequency(collection,
  new String("red")));
```

returns **2**

The **`Vector`** and **`Stack`** Classes

- The Java Collections Framework was introduced with Java 2 (JDK1.2)
  - 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



The `Vector` Class

- **`Vector`** is the same as **`ArrayList`**, except that it contains **`synchronized`** methods for accessing and modifying the vector
  - Synchronized methods can prevent data corruption when a vector is accessed and modified by two or more threads concurrently
    - None of the classes discussed until now are synchronized
  - For many applications that do not require synchronization, using **`ArrayList`** is more efficient than using **`Vector`**
- Method retained from Java 2:
  - **`addElement(Object element)`** is the same as the **`add(Object element)`** method, except that the **`addElement`** method is synchronized

The **Vector** Class

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 **<u>last-in-first-out</u>** stack of objects
  - The elements are accessed only from the top of the stack
    - You can insert with push(o:E), retrieve with peek() and remove with pop() (all from the top of the stack)
  - **`Stack`** is implemented as an extension of **`Vector`**
    - Method retained from Java 2:
      - **`empty()`** method is the same as **`isEmpty()`**

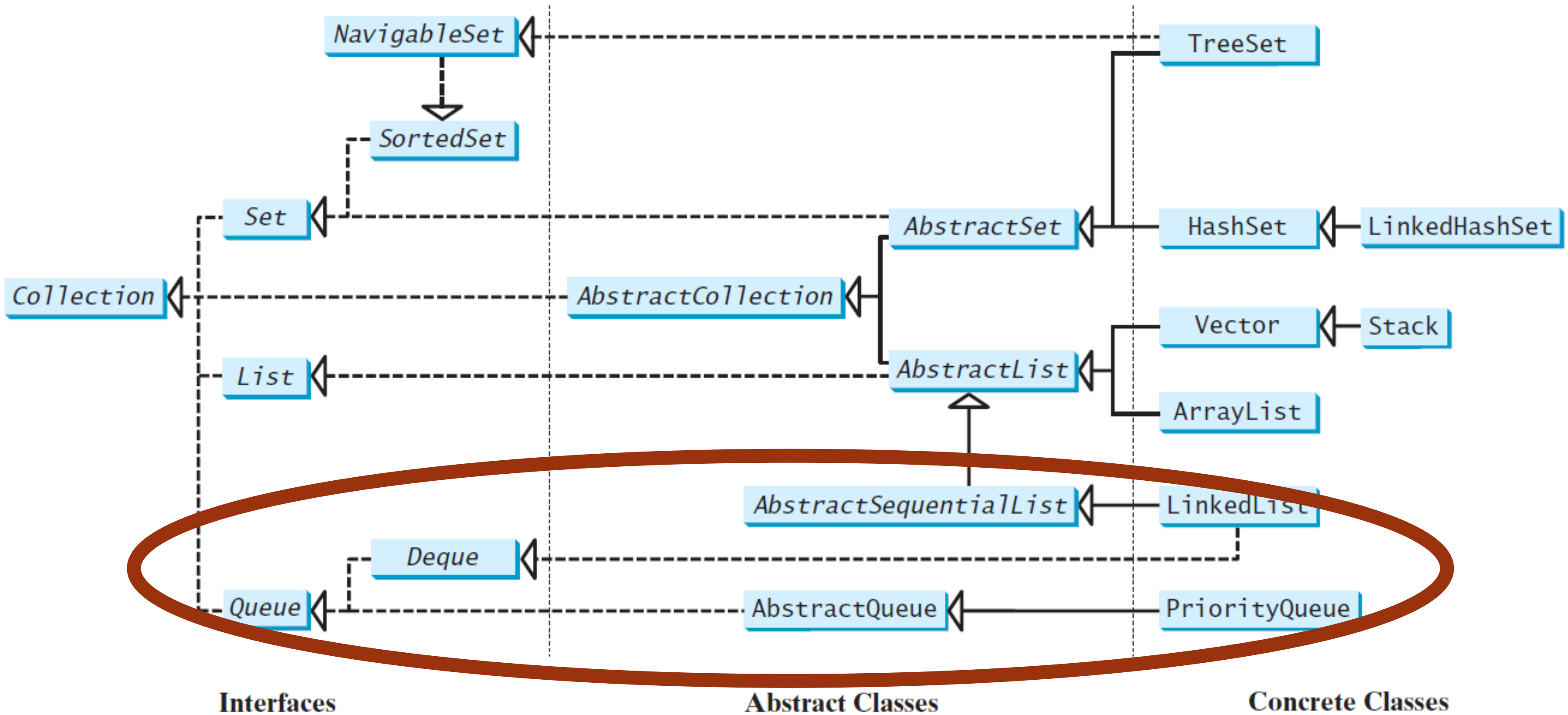
The `Stack` Class

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

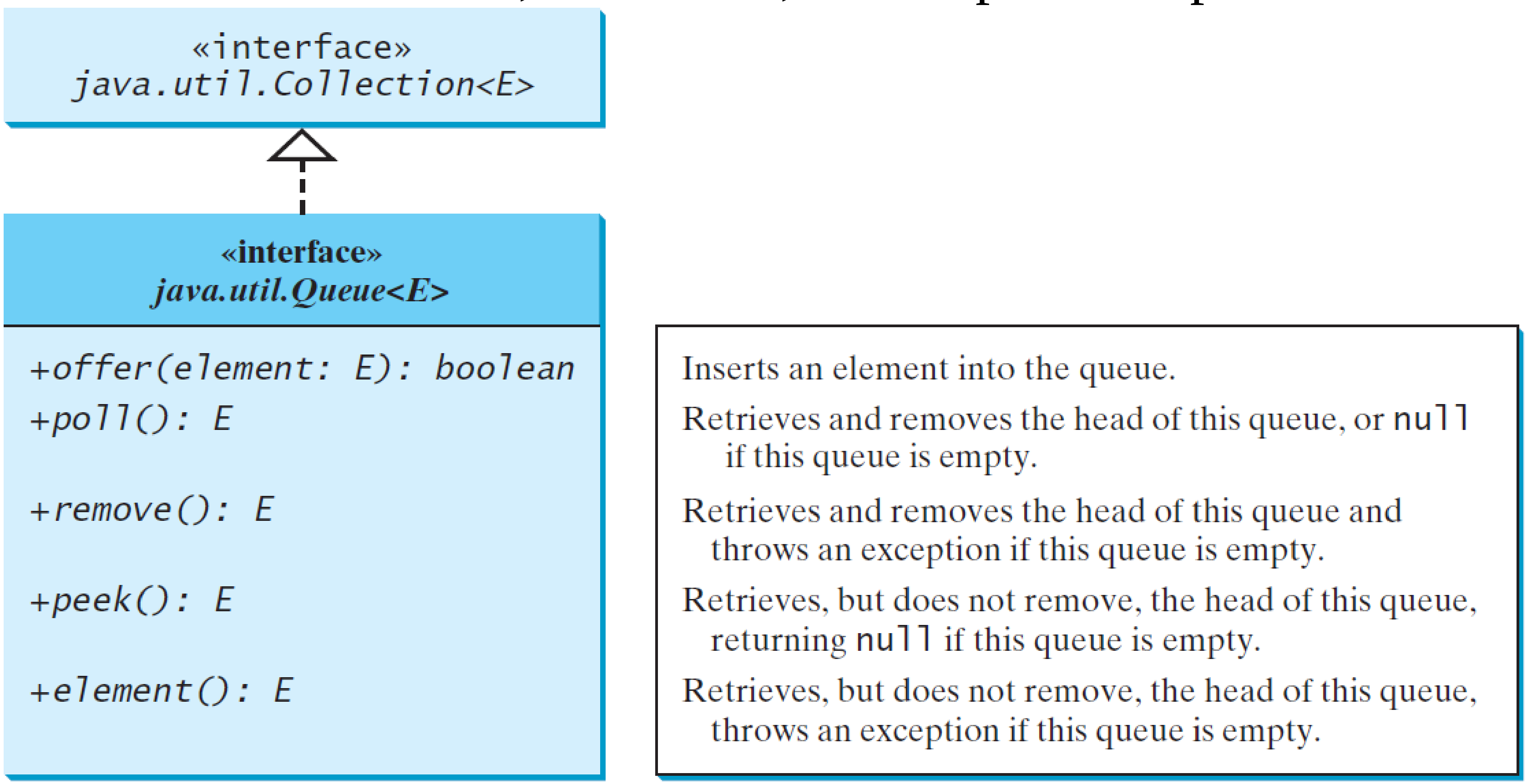


Queues and Priority Queues

- 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
    - The `offer(o:E)` method is used to add an element to the queue
      - This method is similar to the `add` method in the `Collection` interface, but the `offer` method is preferred for queues
    - The `poll` and `remove` methods are similar, except that `poll()` returns `null` if the queue is empty, whereas `remove()` throws an exception
    - The `peek` and `element` methods are similar, except that `peek()` returns `null` if the queue is empty, whereas `element()` throws an exception
- In a ***priority queue***, elements are assigned priorities
  - When accessing elements, the element with the highest priority is removed first

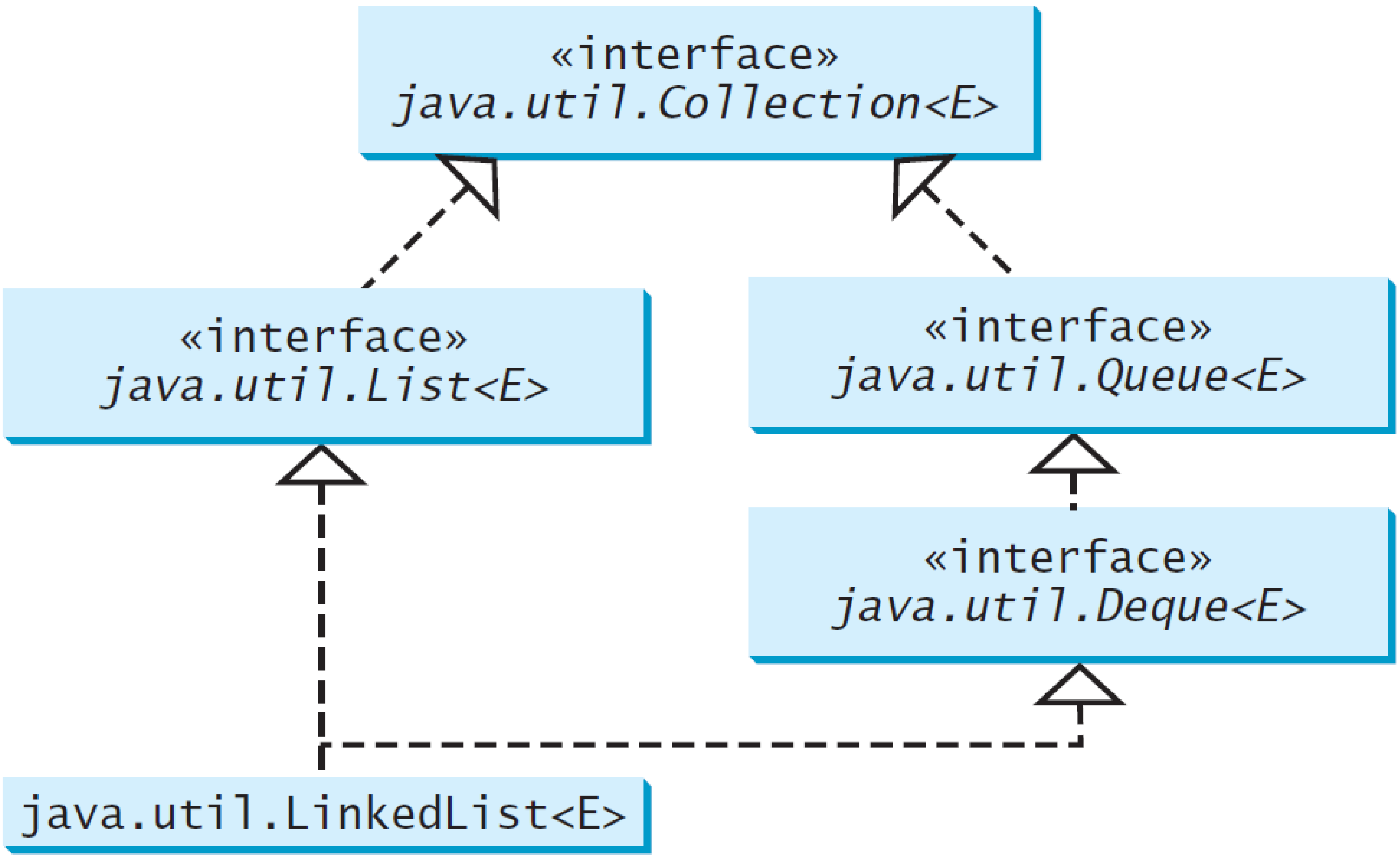
The **Queue** Interface

- **`Queue`** interface extends **`java.util.Collection`** with additional insertion, extraction, and inspection operations



Using **`LinkedList`** for **`Queue`**

- The **`LinkedList`** class implements the **`Deque`** interface, which extends the **`Queue`** interface



Queues

- **`Deque`** interface supports element insertion and removal at both ends
  - The name deque is short for "double-ended queue"
- The **`Deque`** interface extends **`Queue`** with additional methods for inserting and removing elements from both ends of the queue
  - The methods **`addFirst(e)`**, **`removeFirst()`**, **`addLast(e)`**, **`removeLast()`**, **`getFirst()`**, and **`getLast()`** are defined in the **`Deque`** interface

```
java.util.Queue<String> queue =
        new java.util.LinkedList<>();
queue.offer("Oklahoma");
queue.offer("Indiana");
queue.offer("Georgia");
queue.offer("Texas");
while (queue.size() > 0)
        System.out.print(queue.remove() + " ");
```

returns **`Oklahoma Indiana Georgia Texas`**

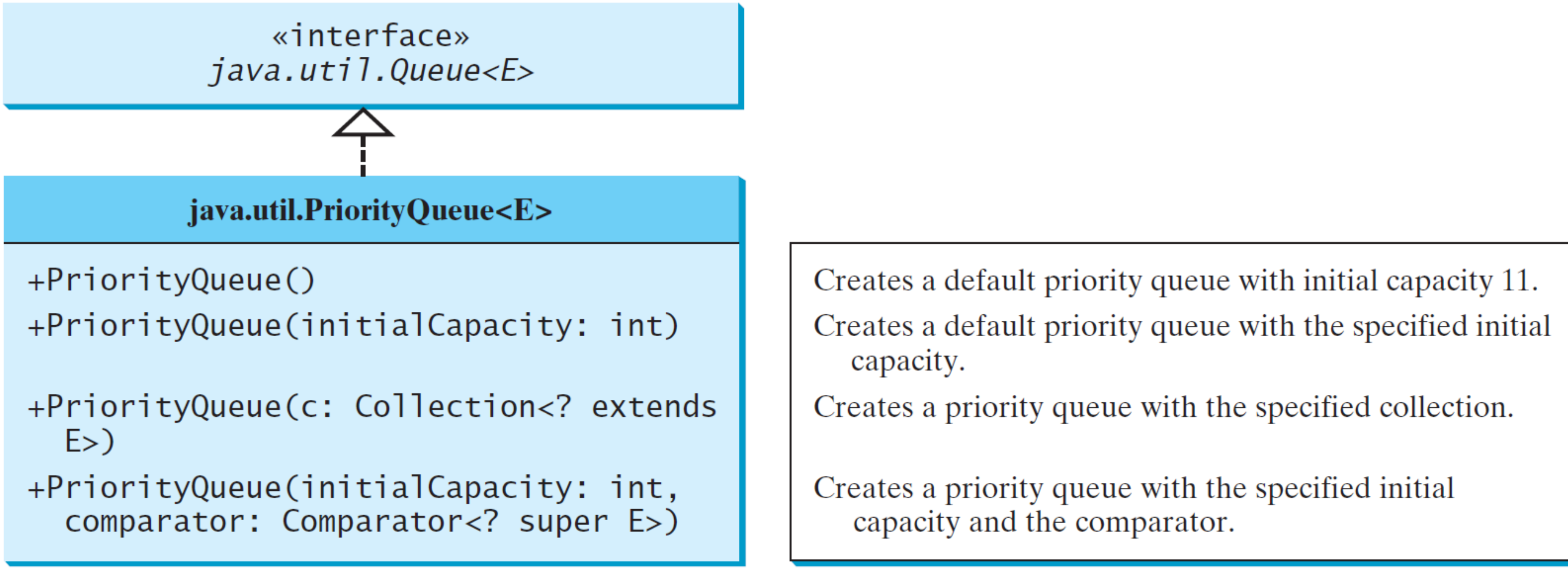
- **`LinkedList`** is the concrete class for queue and it supports inserting and removing elements from both ends of a list

Priority Queues

- **`java.util.PriorityQueue<T>`**
  - By default, the priority queue orders its elements according to their natural ordering using **`Comparable`**
  - The element with the **least value** is assigned the **highest priority** and thus is removed from the queue first
  - If there are several elements with the same highest priority, the tie is broken arbitrarily
  - You can also specify an ordering using **`Comparator`** in the constructor

`PriorityQueue(initialCapacity,comparator)`

The PriorityQueue Class



```java
import java.util.*;
public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<String> queue1 = new PriorityQueue<>();
    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() + " ");
    } // Georgia Indiana Oklahoma Texas

    PriorityQueue<String> queue2 = new PriorityQueue<>(
        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() + " ");
    } // Texas Oklahoma Indiana Georgia
  }
}
```

Case Study: Evaluating Expressions

- Stacks can be used to evaluate expressions

```
c:\book>java EvaluateExpression "(1 + 3 * 3 - 2) * (12 / 6 * 5)"
80

c:\book>java EvaluateExpression "(1 + 3 * 3 - 2) * (12 / 6 * 5) +"
Wrong expression: (1 + 3 * 3 - 2) * (12 / 6 * 5) +

c:\book>java EvaluateExpression "(1 + 2) * 4 - 3"
9

c:\book>
```

Example Stack Algorithm for parsing

- **Phase 1: Scan the expression with infix operators from left to right to extract operands, operators, and the parentheses and compute the value of the expression**
  - 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 on the **`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 Stack Algorithm for parsing

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	

```java
import java.util.Stack;

public class EvaluateExpression {
  public static void main(String[] args) {
    // Check number of arguments passed
    if (args.length != 1) {
      System.out.println(
        "Usage: java EvaluateExpression \"expression\"");
      System.exit(1);
    }

    try {
      System.out.println(evaluateExpression(args[0]));
    }
    catch (Exception ex) {
      System.out.println("Wrong expression: " + args[0]);
    }
  }

  /** Evaluate an expression */
  public static int evaluateExpression(String expression) {
    // Create operandStack to store operands
    Stack<Integer> operandStack = new Stack<>();

    // Create operatorStack to store operators
    Stack<Character> operatorStack = new Stack<>();
```

```
// Insert blanks around (, ), +, -, /, and *
expression = insertBlanks(expression);

// Extract operands and operators
String[] tokens = expression.split(" ");

// Phase 1: Scan tokens
for (String token: tokens) {
  if (token.length() == 0) // Blank space
    continue; // Back to the while loop to extract the next token
  else if (token.charAt(0) == '+' || token.charAt(0) == '-') {
    // Process all +, -, *, / in the top of the operator stack
    while (!operatorStack.isEmpty() &&
      (operatorStack.peek() == '+' ||
       operatorStack.peek() == '-' ||
       operatorStack.peek() == '*' ||
       operatorStack.peek() == '/')) {
     processAnOperator(operandStack, operatorStack);
    }

    // Push the + or - operator into the operator stack
    operatorStack.push(token.charAt(0));
  }
```

```java
    else if (token.charAt(0) == '*' || token.charAt(0) == '/') {
      // Process all *, / in the top of the operator stack
      while (!operatorStack.isEmpty() &&
        (operatorStack.peek() == '*' ||
        operatorStack.peek() == '/')) {
        processAnOperator(operandStack, operatorStack);
      }

      // Push the * or / operator into the operator stack
      operatorStack.push(token.charAt(0));
    } else if (token.trim().charAt(0) == '(') {
      operatorStack.push('('); // Push '(' to stack
    } else if (token.trim().charAt(0) == ')') {
      // Process all the operators in the stack until seeing '('
      while (operatorStack.peek() != '(') {
        processAnOperator(operandStack, operatorStack);
      }

      operatorStack.pop(); // Pop the '(' symbol from the stack
    } else { // An operand scanned
      // Push an operand to the stack
      operandStack.push(new Integer(token));
    }
  }
```

```java
    // Phase 2: process all the remaining operators in the stack
    while (!operatorStack.isEmpty()) {
      processAnOperator(operandStack, operatorStack);
    }

    // Return the result
    return operandStack.pop();
  }

  /** Process one operator: Take an operator from operatorStack and
   *  apply it on the operands in the operandStack */
  public static void processAnOperator(
      Stack<Integer> operandStack, Stack<Character> operatorStack) {
    char op = operatorStack.pop();
    int op1 = operandStack.pop();
    int op2 = operandStack.pop();
    if (op == '+')
      operandStack.push(op2 + op1);
    else if (op == '-')
      operandStack.push(op2 - op1);
    else if (op == '*')
      operandStack.push(op2 * op1);
    else if (op == '/')
      operandStack.push(op2 / op1);
  }
```

```
  public static String insertBlanks(String s) {
    String result = "";

    for (int i = 0; i < s.length(); i++) {
      if (s.charAt(i) == '(' || s.charAt(i) == ')' ||
          s.charAt(i) == '+' || s.charAt(i) == '-' ||
          s.charAt(i) == '*' || s.charAt(i) == '/')
        result += " " + s.charAt(i) + " ";
      else
        result += s.charAt(i);
    }

    return result;
  }
}
```