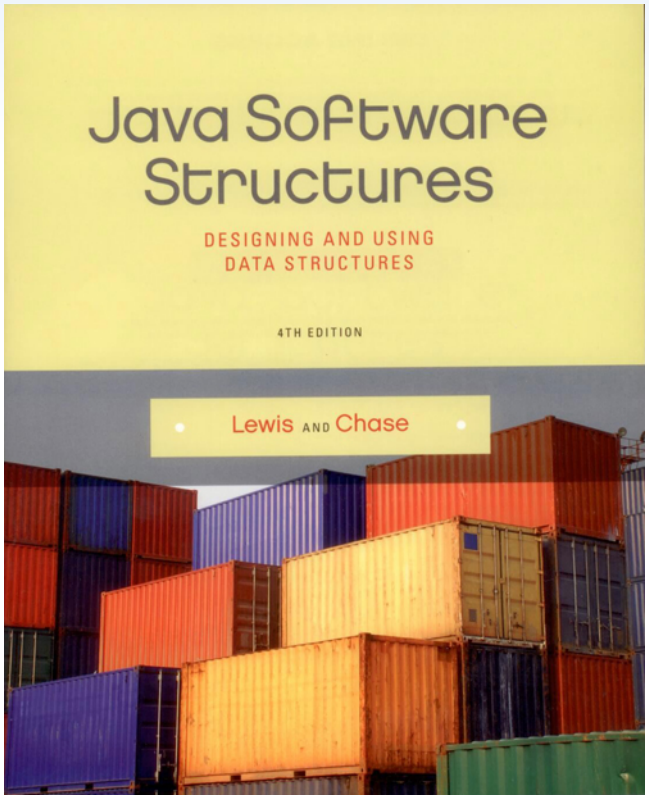


Chapter 6

Lists

Chapter Scope

- Types of list collections
- Using lists to solve problems
- Various list implementations
- Comparing list implementations

Lists

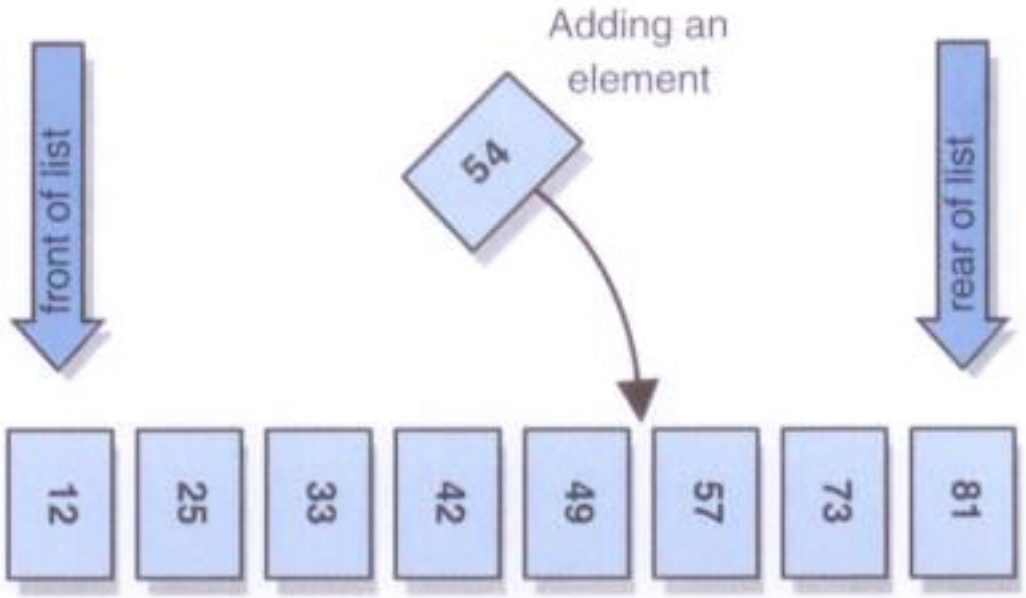
- A list is a linear collection, like stacks and queues, but is more flexible
- Adding and removing elements in lists can occur at either end or anywhere in the middle
- We will examine three types of list collections:
  - ordered lists
  - unordered lists
  - indexed lists

Ordered Lists

- The elements in an *ordered list* are ordered by some inherent characteristic of the elements
  - names in alphabetical order
  - scores in ascending order
- The elements themselves determine where they are stored in the list

Ordered Lists

- An ordered list:

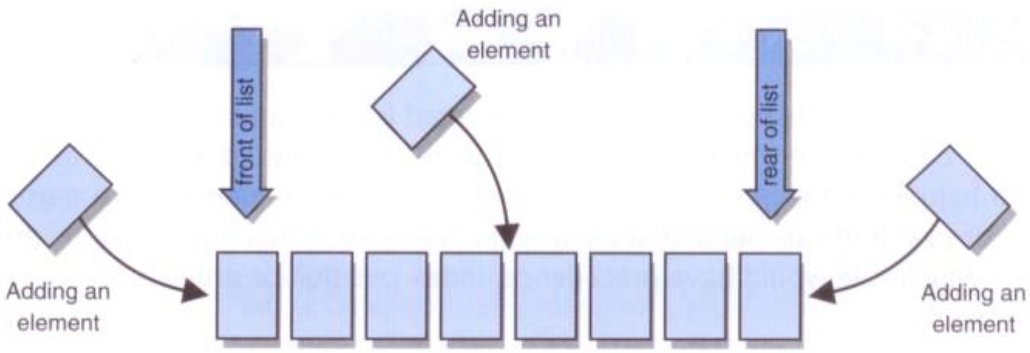


Unordered Lists

- There is an order to the elements in an *unordered list*, but that order is not based on element characteristics
- The user of the list determines the order of the elements
- A new element can be put on the front or the rear of the list, or it can be inserted after a particular element already in the list

Unordered Lists

- An unordered list:

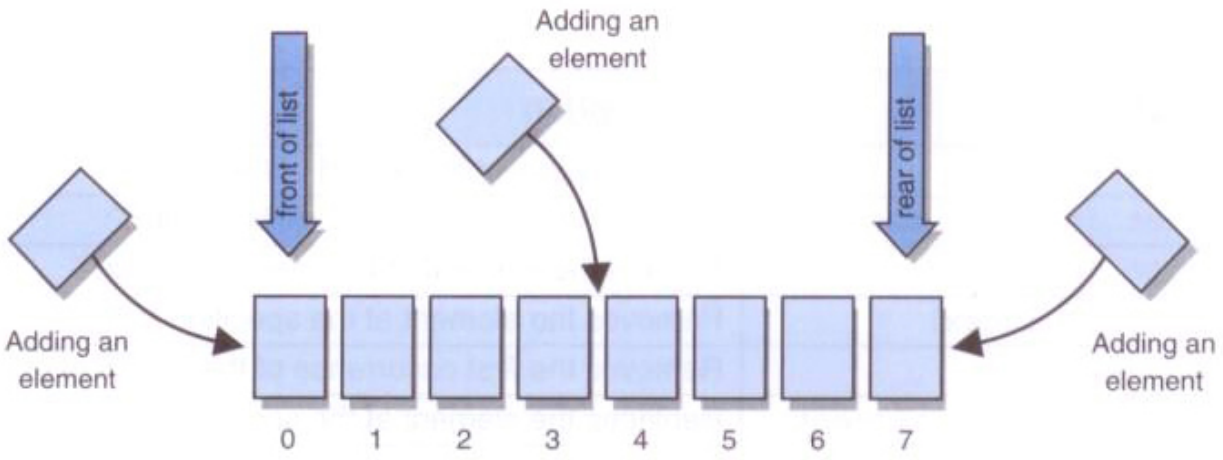


Indexed Lists

- In an *indexed list*, elements are referenced by their numeric position in the list
- Like an unordered list, there is no inherent relationship among the elements
- The user can determine the order
- Every time the list changes, the indexes are updated

Indexed Lists

- An indexed list:



Lists in the Java API

- The list classes in the Java API primarily support the concept of an indexed list (and somewhat an unordered list)
- The API does not have any classes that directly implement an ordered list
- The `ArrayList` and `LinkedList` classes both implement the `List<E>` interface

Lists in the Java API

- Some of the operations from the `List<E>` interface:

Method	Description
`add(E element)`	Adds an element to the end of the list.
`add(int index, E element)`	Inserts an element at the specified index.
`get(int index)`	Returns the element at the specified index.
`remove(int index)`	Removes the element at the specified index.
`remove(E object)`	Removes the first occurrence of the specified object.
`set(int index, E element)`	Replaces the element at the specified index.
`size()`	Returns the number ofelements in the list.

Program of Study

- Let's use an unordered list to manage the courses that a student takes to fulfill degree requirements
- The `Course` class represents a course, which may or may not have already been taken
- The `ProgramOfStudy` course manages a list of `Course` objects
- The list is stored for later use using *serialization*

```
import java.io.IOException;

/**
 * Demonstrates the use of a list to manage a set of objects.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class POSTester
{
    /**
     * Creates and populates a Program of Study. Then saves it using serialization.
     */
    public static void main(String[] args) throws IOException
    {
     ProgramOfStudy pos = new ProgramOfStudy();

     pos.addCourse(new Course("CS", 101, "Introduction to Programming", "A-"));
     pos.addCourse(new Course("ARCH", 305, "Building Analysis", "A"));
     pos.addCourse(new Course("GER", 210, "Intermediate German"));
     pos.addCourse(new Course("CS", 320, "Computer Architecture"));
     pos.addCourse(new Course("THE", 201, "The Theatre Experience"));

     Course arch = pos.find("CS", 320);
     pos.addCourseAfter(arch, new Course("CS", 321, "Operating Systems"));
```

```
      Course theatre = pos.find("THE", 201);
      theatre.setGrade("A-");

      Course german = pos.find("GER", 210);
      pos.replace(german, new Course("FRE", 110, "Beginning French", "B+"));

      System.out.println(pos);

      pos.save("ProgramOfStudy");
    }
}
```

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

/**
 * Represents a Program of Study, a list of courses taken and planned, for an
 * individual student.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class ProgramOfStudy implements Iterable<Course>, Serializable
{
    private List<Course> list;

    /**
     * Constructs an initially empty Program of Study.
     */
    public ProgramOfStudy()
    {
      list = new LinkedList<Course>();
    }
```

```
/**
 * Adds the specified course to the end of the course list.
 *
 * @param course the course to add
 */
public void addCourse(Course course)
{
  if (course != null)
        list.add(course);
}

/**
 * Finds and returns the course matching the specified prefix and number.
 *
 * @param prefix the prefix of the target course
 * @param number the number of the target course
 * @return the course, or null if not found
 */
public Course find(String prefix, int number)
{
  for (Course course : list)
        if (prefix.equals(course.getPrefix()) &&
                  number == course.getNumber())
              return course;

  return null;
}
```

```
/**
 * Adds the specified course after the target course. Does nothing if
 * either course is null or if the target is not found.
 *
 * @param target the course after which the new course will be added
 * @param newCourse the course to add
 */
public void addCourseAfter(Course target, Course newCourse)
{
 if (target == null || newCourse == null)
      return;

 int targetIndex = list.indexOf(target);
 if (targetIndex != -1)
      list.add(targetIndex + 1, newCourse);
}
```

```
/**
 * Replaces the specified target course with the new course. Does nothing if
 * either course is null or if the target is not found.
 *
 * @param target the course to be replaced
 * @param newCourse the new course to add
 */
public void replace(Course target, Course newCourse)
{
 if (target == null || newCourse == null)
       return;

 int targetIndex = list.indexOf(target);
 if (targetIndex != -1)
       list.set(targetIndex, newCourse);
}

/**
 * Creates and returns a string representation of this Program of Study.
 *
 * @return a string representation of the Program of Study
 */
public String toString()
{
 String result = "";
 for (Course course : list)
       result += course + "\n";
 return result;
}
```

```
/**
 * Returns an iterator for this Program of Study.
 *
 * @return an iterator for the Program of Study
 */
public Iterator<Course> iterator()
{
 return list.iterator();
}

/**
 * Saves a serialized version of this Program of Study to the specified
 * file name.
 *
 * @param fileName the file name under which the POS will be stored
 * @throws IOException
 */
public void save(String fileName) throws IOException
{
 FileOutputStream fos = new FileOutputStream(fileName);
 ObjectOutputStream oos = new ObjectOutputStream(fos);
 oos.writeObject(this);
 oos.flush();
 oos.close();
}
```

```
    /**
     * Loads a serialized Program of Study from the specified file.
     *
     * @param fileName the file from which the POS is read
     * @return the loaded Program of Study
     * @throws IOException
     * @throws ClassNotFoundException
     */
    public static ProgramOfStudy load(String fileName) throws IOException,
    ClassNotFoundException
    {
      FileInputStream fis = new FileInputStream(fileName);
      ObjectInputStream ois = new ObjectInputStream(fis);
      ProgramOfStudy pos = (ProgramOfStudy) ois.readObject();
      ois.close();

      return pos;
    }
}
```

```
import java.io.Serializable;

/**
 * Represents a course that might be taken by a student.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Course implements Serializable
{
    private String prefix;
    private int number;
    private String title;
    private String grade;
```

```
/**
 * Constructs the course with the specified information.
 *
 * @param prefix the prefix of the course designation
 * @param number the number of the course designation
 * @param title the title of the course
 * @param grade the grade received for the course
 */
public Course(String prefix, int number, String title, String grade)
{
  this.prefix = prefix;
  this.number = number;
  this.title = title;
  if (grade == null)
        this.grade = "";
  else
        this.grade = grade;
}

/**
 * Constructs the course with the specified information, with no grade
 * established.
 *
 * @param prefix the prefix of the course designation
 * @param number the number of the course designation
 * @param title the title of the course
 */
public Course(String prefix, int number, String title)
{
  this(prefix, number, title, "");
}
```

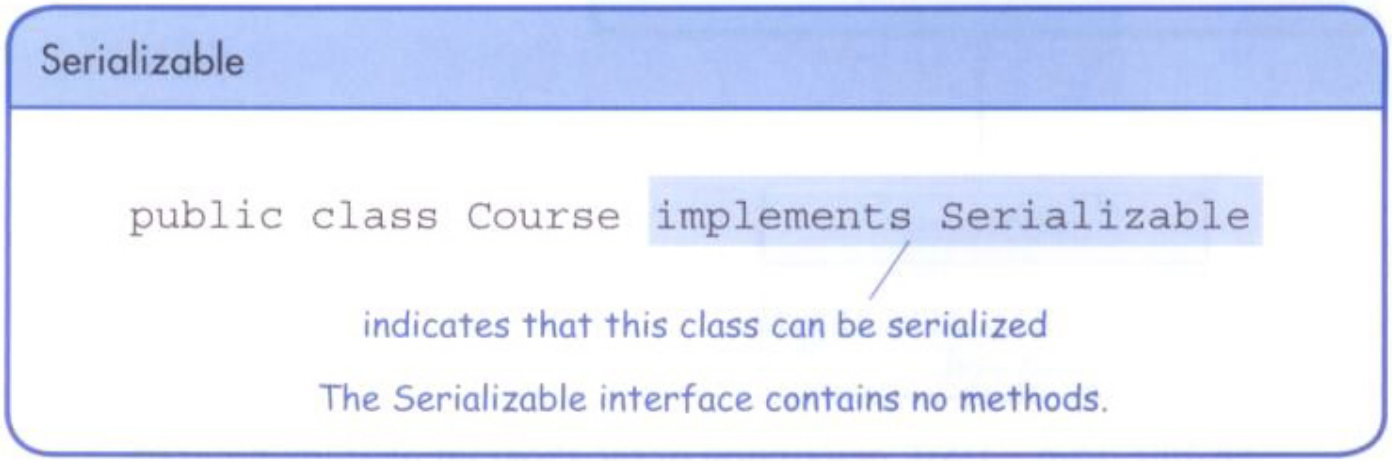
```
/**
 * Returns the prefix of the course designation.
 *
 * @return the prefix of the course designation
 */
public String getPrefix()
{
  return prefix;
}

/**
 * Returns the number of the course designation.
 *
 * @return the number of the course designation
 */
public int getNumber()
{
  return number;
}

/**
 * Returns the title of this course.
 *
 * @return the prefix of the course
 */
public String getTitle()
{
  return title;
}
```

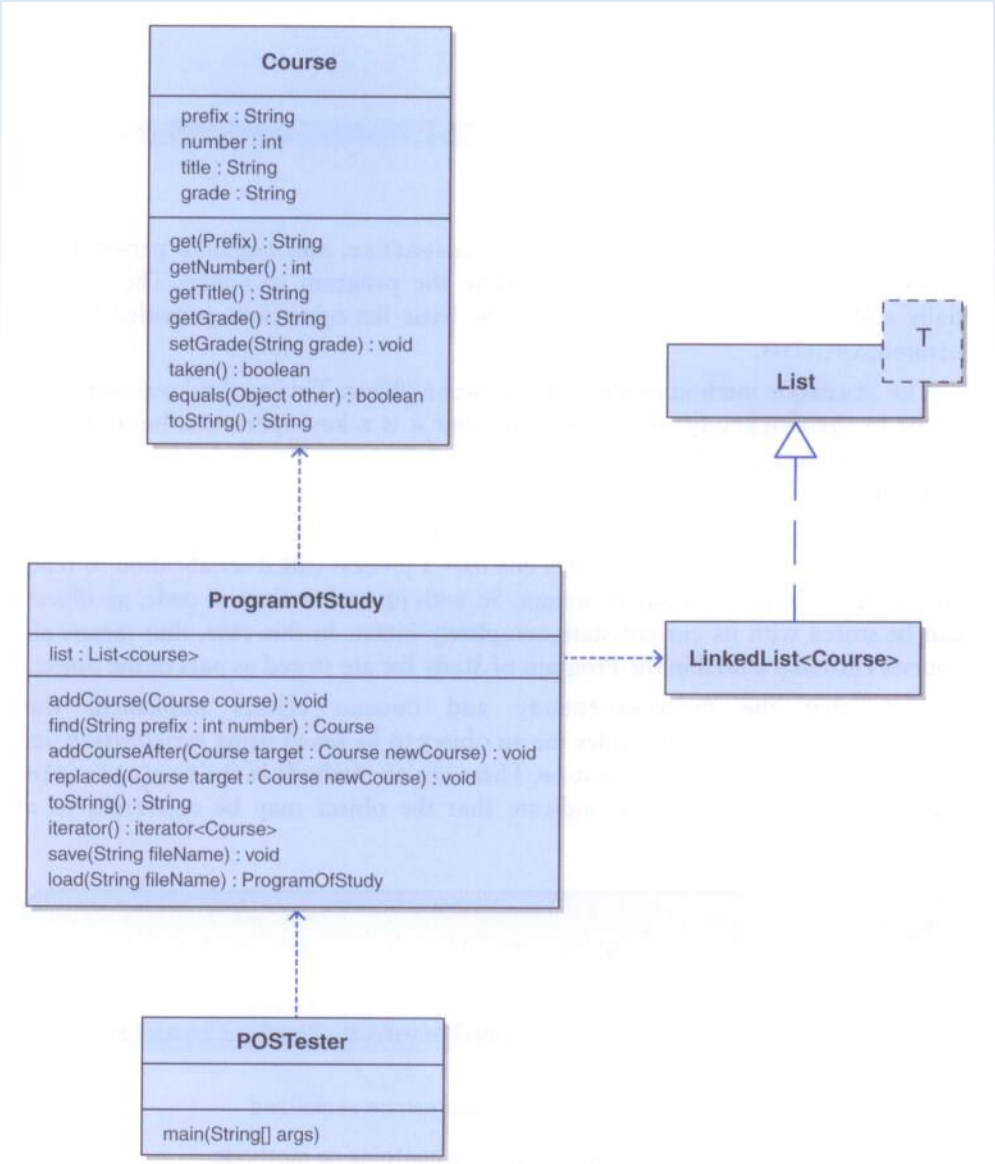
```
/**
 * Returns the grade for this course.
 *
 * @return the grade for this course
 */
public String getGrade()
{
  return grade;
}

/**
 * Sets the grade for this course to the one specified.
 *
 * @param grade the new grade for the course
 */
public void setGrade(String grade)
{
  this.grade = grade;
}

/**
 * Returns true if this course has been taken (if a grade has been received).
 *
 * @return true if this course has been taken and false otherwise
 */
public boolean taken()
{
  return !grade.equals("");
}
```

```
    /**
     * Determines if this course is equal to the one specified, based on the
     * course designation (prefix and number).
     *
     * @return true if this course is equal to the parameter
     */
    public boolean equals(Object other)
    {
      boolean result = false;
      if (other instanceof Course)
      {
            Course otherCourse = (Course) other;
            if (prefix.equals(otherCourse.getPrefix()) &&
                        number == otherCourse.getNumber())
                  result = true;
      }
      return result;
    }

    /**
     * Creates and returns a string representation of this course.
     *
     * @return a string representation of the course
     */
    public String toString()
    {
      String result = prefix + " " + number + ": " + title;
      if (!grade.equals(""))
            result += "  [" + grade + "]";
      return result;
    }
}
```

Serialization

- Any class whose objects will be saved are tagged with the `Serializable` interface




Course
prefix : String
number : int
title : String
grade : String
get(Prefix) : String
getNumber() : int
getTitle() : String
getGrade() : String
setGrade(String grade) : void
taken() : boolean
equals(Object other) : boolean
toString() : String
List
T
ProgramOfStudy
list : List<course>
addCourse(Course course) : void
find(String prefix : int number) : Course
addCourseAfter(Course target : Course newCourse) : void
replaced(Course target : Course newCourse) : void
toString() : String
iterator() : iterator<Course>
save(String fileName) : void
load(String fileName) : ProgramOfStudy
LinkedList<Course>
POSTester
main(String[] args)

The Josephus Problem

- In a *Josephus problem*, a set of elements are arranged in a circle
- Starting with a particular element, every i^{th} element is removed
- Processing continues until there is only one element left
- The question: given the starting point and remove count (i), which element is left?

```
import java.util.*;

/**
 * Demonstrates the use of an indexed list to solve the Josephus problem.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Josephus
{
    /**
     * Continue around the circle eliminating every nth soldier
     * until all of the soldiers have been eliminated.
     */
    public static void main(String[] args)
    {
        int numPeople, skip, targetIndex;
        List<String> list = new ArrayList<String>();
        Scanner in = new Scanner(System.in);

        // get the initial number of soldiers
        System.out.print("Enter the number of soldiers: ");
        numPeople = in.nextInt();
        in.nextLine();

        // get the number of soldiers to skip
        System.out.print("Enter the number of soldiers to skip: ");
        skip = in.nextInt();
```

```
        // load the initial list of soldiers
        for (int count = 1; count <= numPeople; count++)
        {
            list.add("Soldier " + count);
        }

        targetIndex = skip;
        System.out.println("The order is: ");

        // Treating the list as circular, remove every nth element
        // until the list is empty
        while (!list.isEmpty())
        {
            System.out.println(list.remove(targetIndex));
            if (list.size() > 0)
                targetIndex = (targetIndex + skip) % list.size();
        }
    }
}
```

Implementing Lists

- Now let's implement our own list collection
- The following operations are common to all types of lists:

Operation	Description
`removeFirst`	Removes the first element from the list.
`removeLast`	Removes the last element from the list.
`remove`	Removes a particular element from the list.
`first`	Examines the element at the front of the list.
`last`	Examines the element at the rear of the list.
`contains`	Determines if the list contains a particular element.
`isEmpty`	Determines if the list is empty.
`size`	Determines the number of elements on the list.

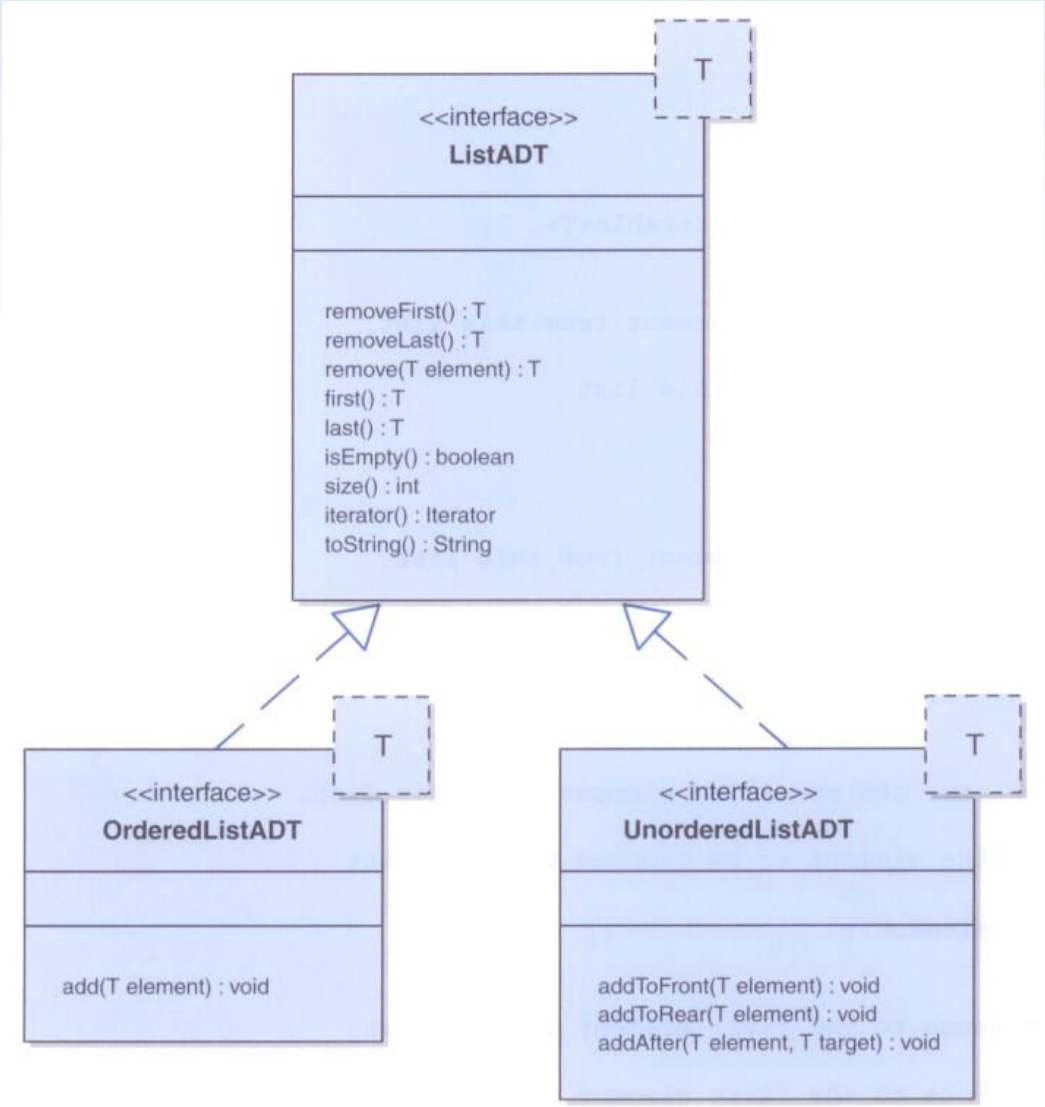
Implementing Lists

- Operation particular to an ordered list:

Operation	Description
`add`	Adds an element to the list.

- Operations particular to an unordered lists:

Operation	Description
`addToFront`	Adds an element to the front of the list.
`addToRear`	Adds an element to the rear of the list.
`addAfter`	Adds an element after a particular element already in the list.


T
<<interface>>
ListADT
removeFirst() : T
removeLast() : T
remove(T element) : T
first() : T
last() : T
isEmpty() : boolean
size() : int
iterator() : Iterator
toString() : String
T
<<interface>>
OrderedListADT
add(T element) : void
T
<<interface>>
UnorderedListADT
addToFront(T element) : void
addToRear(T element) : void
addAfter(T element, T target) : void

```
package jsjf;

import java.util.Iterator;

/**
 * ListADT defines the interface to a general list collection. Specific
 * types of lists will extend this interface to complete the
 * set of necessary operations.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface ListADT<T> extends Iterable<T>
{
    /**
     * Removes and returns the first element from this list.
     *
     * @return the first element from this list
     */
    public T removeFirst();

    /**
     * Removes and returns the last element from this list.
     *
     * @return the last element from this list
     */
    public T removeLast();
```

```
/**
 * Removes and returns the specified element from this list.
 *
 * @param element the element to be removed from the list
 */
public T remove(T element);

/**
 * Returns a reference to the first element in this list.
 *
 * @return a reference to the first element in this list
 */
public T first();

/**
 * Returns a reference to the last element in this list.
 *
 * @return a reference to the last element in this list
 */
public T last();

/**
 * Returns true if this list contains the specified target element.
 *
 * @param target the target that is being sought in the list
 * @return true if the list contains this element
 */
public boolean contains(T target);
```

```
    /**
     * Returns true if this list contains no elements.
     *
     * @return true if this list contains no elements
     */
    public boolean isEmpty();

    /**
     * Returns the number of elements in this list.
     *
     * @return the integer representation of number of elements in this list
     */
    public int size();

    /**
     * Returns an iterator for the elements in this list.
     *
     * @return an iterator over the elements in this list
     */
    public Iterator<T> iterator();

    /**
     * Returns a string representation of this list.
     *
     * @return a string representation of this list
     */
    public String toString();
}
```

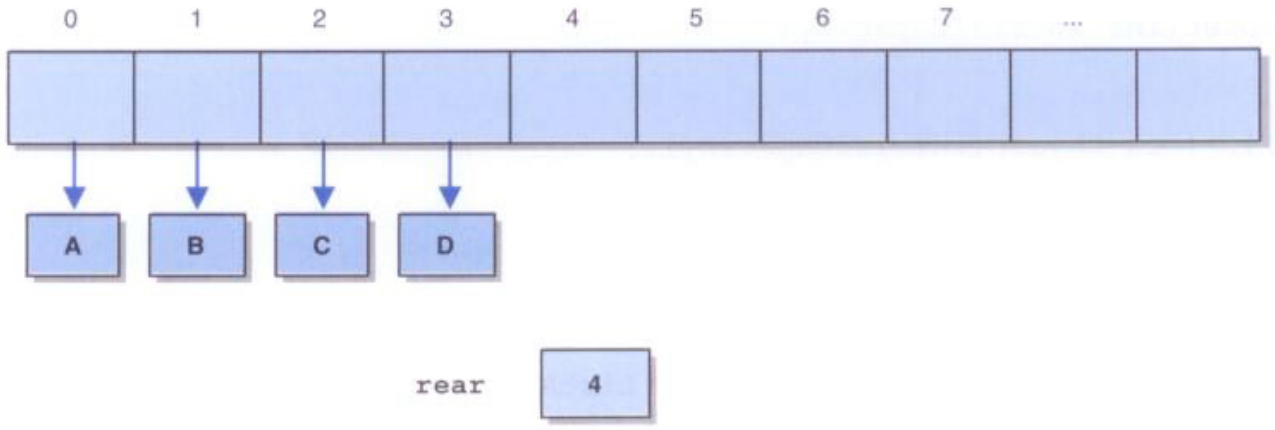
```
package jsjf;

/**
 * OrderedListADT defines the interface to an ordered list collection. Only
 * Comparable elements are stored, kept in the order determined by
 * the inherent relationship among the elements.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface OrderedListADT<T> extends ListADT<T>
{
    /**
     * Adds the specified element to this list at the proper location
     *
     * @param element the element to be added to this list
     */
    public void add(T element);
}
```

```
package jsjf;

/**
 * UnorderedListADT defines the interface to an unordered list collection.
 * Elements are stored in any order the user desires.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface UnorderedListADT<T> extends ListADT<T>
{
    /**
     * Adds the specified element to the front of this list.
     *
     * @param element the element to be added to the front of this list
     */
    public void addToFront(T element);
```

```
    /**
     * Adds the specified element to the rear of this list.
     *
     * @param element the element to be added to the rear of this list
     */
    public void addToRear(T element);

    /**
     * Adds the specified element after the specified target.
     *
     * @param element the element to be added after the target
     * @param target  the target is the item that the element will be added
     *                after
     */
    public void addAfter(T element, T target);
}
```

Implementing a List with an Array

- Since elements can be added anywhere in the list, shifting elements cannot be avoided
- So a straightforward implementation can be adopted:



```
package jsjf;

import jsjf.exceptions.*;
import java.util.*;

/**
 * ArrayList represents an array implementation of a list. The front of
 * the list is kept at array index 0. This class will be extended
 * to create a specific kind of list.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public abstract class ArrayList<T> implements ListADT<T>, Iterable<T>
{
    private final static int DEFAULT_CAPACITY = 100;
    private final static int NOT_FOUND = -1;

    protected int rear;
    protected T[] list;
    protected int modCount;

    /**
     * Creates an empty list using the default capacity.
     */
    public ArrayList()
    {
        this(DEFAULT_CAPACITY);
    }
```

```
/**
 * Creates an empty list using the specified capacity.
 *
 * @param initialCapacity the integer value of the size of the array list
 */
public ArrayList(int initialCapacity)
{
    rear = 0;
    list = (T[])(new Object[initialCapacity]);
    modCount = 0;
}
```

```
/**
 * Removes and returns the specified element.
 *
 * @param  element the element to be removed and returned from the list
 * @return the removed elememt
 * @throws ElementNotFoundException if the element is not in the list
 */
public T remove(T element)
{
    T result;
    int index = find(element);

    if (index == NOT_FOUND)
        throw new ElementNotFoundException("ArrayList");

    result = list[index];
    rear--;

    // shift the appropriate elements
    for (int scan=index; scan < rear; scan++)
        list[scan] = list[scan+1];

    list[rear] = null;
    modCount++;

    return result;
}
```

```
/**
 * Returns true if this list contains the specified element.
 *
 * @param target the target element
 * @return true if the target is in the list, false otherwise
 */
public boolean contains(T target)
{
    return (find(target) != NOT_FOUND);
}

/**
 * Returns the array index of the specified element, or the
 * constant NOT_FOUND if it is not found.
 *
 * @param target the target element
 * @return the index of the target element, or the
 *         NOT_FOUND constant
 */
private int find(T target)
{
    int scan = 0;
  int result = NOT_FOUND;

    if (!isEmpty())
        while (result == NOT_FOUND && scan < rear)
            if (target.equals(list[scan]))
                result = scan;
            else
                scan++;

    return result;
}
```

```
/**
 * Adds the specified Comparable element to this list, keeping
 * the elements in sorted order.
 *
 * @param element the element to be added to the list
 */
public void add(T element)
{
  if (!(element instanceof Comparable))
        throw new NonComparableElementException("OrderedList");

  Comparable<T> comparableElement = (Comparable<T>)element;

  if (size() == list.length)
        expandCapacity();

    int scan = 0;

    // find the insertion location
    while (scan < rear && comparableElement.compareTo(list[scan]) > 0)
        scan++;

    // shift existing elements up one
    for (int shift=rear; shift > scan; shift--)
        list[shift] = list[shift-1];

    // insert element
    list[scan] = element;
    rear++;
   modCount++;
}
```

```
/**
 * Adds the specified element after the specified target element.
 * Throws an ElementNotFoundException if the target is not found.
 *
 * @param element the element to be added after the target element
 * @param target  the target that the element is to be added after
 */
public void addAfter(T element, T target)
{
    if (size() == list.length)
        expandCapacity();

    int scan = 0;

    // find the insertion point
    while (scan < rear && !target.equals(list[scan]))
        scan++;

    if (scan == rear)
        throw new ElementNotFoundException("UnorderedList");

    scan++;

    // shift elements up one
    for (int shift=rear; shift > scan; shift--)
        list[shift] = list[shift-1];

    // insert element
    list[scan] = element;
    rear++;
    modCount++;
}
```

Implementing a List with Links

- A classic linked list is an obvious choice for implementing a list collection
- The `LinearNode` class introduced earlier is reused here
- Both `head` and `tail` references are maintained, as well as an integer `count`

```
package jsjf;

import jsjf.exceptions.*;
import java.util.*;

/**
 * LinkedList represents a linked implementation of a list.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public abstract class LinkedList<T> implements ListADT<T>, Iterable<T>
{
    protected int count;
    protected LinearNode<T> head, tail;
    protected int modCount;

    /**
     * Creates an empty list.
     */
    public LinkedList()
    {
        count = 0;
        head = tail = null;
        modCount = 0;
    }
```

```
/**
 * Removes the first instance of the specified element from this
 * list and returns a reference to it. Throws an EmptyCollectionException
 * if the list is empty. Throws a ElementNotFoundException if the
 * specified element is not found in the list.
 *
 * @param  targetElement the element to be removed from the list
 * @return a reference to the removed element
 * @throws EmptyCollectionException if the list is empty
 * @throws ElementNotFoundException if the target element is not found
 */
public T remove(T targetElement) throws EmptyCollectionException,
     ElementNotFoundException
{
    if (isEmpty())
        throw new EmptyCollectionException("LinkedList");

    boolean found = false;
    LinearNode<T> previous = null;
    LinearNode<T> current = head;

    while (current != null && !found)
        if (targetElement.equals(current.getElement()))
            found = true;
        else
        {
            previous = current;
            current = current.getNext();
        }
```

```
        if (!found)
            throw new ElementNotFoundException("LinkedList");

        if (size() == 1)  // only one element in the list
            head = tail = null;
        else if (current.equals(head))  // target is at the head
            head = current.getNext();
        else if (current.equals(tail))  // target is at the tail
        {
            tail = previous;
            tail.setNext(null);
        }
        else  // target is in the middle
            previous.setNext(current.getNext());

        count--;
        modCount++;

        return current.getElement();
    }
```