11

Some Fundamental Algorithms for Arrays: Searching and Sorting

INTRODUCTION—BUILDING, SEARCHING, AND SORTING AS TYPICAL OPERATIONS PERFORMED WITH ARRAYS

In Chapter 10, we introduced the array data structure and gave several examples: an array of `int`s, arrays of `String`s, an array of `Card` objects. In this chapter we will describe a class that uses an array: a class `NameList` that maintains a list of `Name` objects. What distinguishes a list from an array? A list object maintains a counter that keeps track of the number of objects in the array. The `NameList` class will have two instance variables: an array of `Name` objects and an `int` variable that keeps track of the number of `Name` objects currently in the list. We could pretend that this list of names is a list of invitees to a party. We might also imagine looking through the list to find a particular name. That action is called *searching* the list. If our list were long enough, it would be a tough job to find a particular name if the elements in the list were in some random order. Putting the list in alphabetical order could be enormously helpful before searching for any particular name in the list. Rearranging a list so that its elements are in some prescribed order is called *sorting* the list.

When we add an element to the list, it is typical simply to put the new element at the end of the list, disregarding its correct position with respect to the names already in the list. However, we might want to build the list from scratch so that it is in order after each new element is added. This kind of list building removes the need to sort the list. However, there may be a trade off in terms of efficiency. It may be that it takes more effort to maintain the list in order than to simply add items to the end of the list and periodically sort the list. We will discuss each of the actions described above in this chapter. Since we know that we will be dealing with names and alphabetical order will be an issue, we will first develop a `Name` class for elements of our list.

The Name Class

Instance Variables and Constructors

We usually think of a person's name as having three distinct parts: the first name, the middle name, and the family name. Each part of the name is easily represented by a `String` object. Hence, our `Name` class will have three instance variables of class `String`:

```
public class Name {
        private String myFirst; // reference to the first name
        private String myMiddle; // reference to the middle name
        private String myFamily; // reference to the family name
```

It makes sense to have a constructor that has three parameters corresponding to each of the three `String`s. The following constructor is similar to the constructor we used for the `Card` class in Chapter 9, except that all parameters are `String`s this time.

```
public Name (String theFirst, String theMiddle, String theFamily) {
    myFirst = theFirst;
    myMiddle = theMiddle;
    myFamily = theFamily;
}// 3 parameter constructor
```

Some people do not have middle names. For those names, let us have a two parameter constructor that defaults the middle name field to an empty `String`.

```
public Name (String theFirst, String theFamily) {
    myFirst = theFirst;
    myMiddle = "";
    myFamily = theFamily;
}// 2 parameter constructor
```

Access and Modifier Methods for the Instance Variables

We should write `get` and `set` methods for the three `String` instance variables. We will write the methods for the `myFirst` instance variable and leave the others as an exercise. A typical access method has no parameters and returns a reference to the object that is the instance variable. Hence, we have the following access or "`get`" method for the `myFirst` instance variable.

```
public String getFirst() {
        return myFirst;
}//getFirst
```

A typical modifier method has a parameter of the same class as the instance variable, has return type `void`, and simply performs an assignment of the parameter to the appropriate instance variable. Hence, we have the following modifier or "`set`" method for the `myFirst` instance variable.

```
public void setFirst(String theFirst) {
        myFirst = theFirst;
}//setFirst
```

Exercise 11.1: Write access and modifier methods for the myMiddle and myFamily instance variables of the Name class.

Methods of the String Class to Assist in Comparing Name Objects

We know that we will be constructing an array of Name objects and doing the work to alphabetize the list. Hence, we will need some way to determine if one Name is before another in alphabetical order. If two Names are compared and the myFamily instance variables are different, we can easily determine which Name comes first in alphabetical order. For example, Li Santi comes before Mann. If, however, the myFamily instance variables are identical, we need to compare the myFirst instance variables of the Names to determine which name comes first in alphabetical order. For example, George Mann comes before Lydia Mann. In the less likely circumstance that two Names match in both the myFamily instance variables and the myFirst instance variables, we would need to compare the myMiddle instance variables to determine which Name comes first in alphabetical order. For example, George Millard Mann comes before George Richard Mann in alphabetical order.

The String class provides us with two methods to assist in this situation, the equals method and the compareTo method. Using the relational operator == compares references, not the contents of the objects to which they refer. Using == with references is essentially asking if the two references point to the same address for an object in memory. In most cases, we need to know if two distinct objects can be considered as having the same data stored in their instance variables. Hence, we usually do not use == when we compare objects.

The equals method in the String class is used as follows:

```
<invoking String>.equals (<argument String>)
```

The equals method of the String class returns true if the invoking String and the argument String have the same length and characters. The equals method of the String class returns false if the invoking String and the argument String are not identical. The signature of the equals method in the String class is equals (Object). Because the Object class is the super class of the String class, any String is an Object. When the equals method of the String class is invoked with an argument that is a reference to a String object, the semantics of parameter passing are the same as assignment. In other words, it is as though the argument String is assigned to the parameter Object. The equals method of the String class is written in this way so that it will override the equals method of the Object class that is inherited by the String class. If the equals method of the String class had instead a String parameter, its signature would not match the equals method of the Object class and the equals method of the Object class would not be overridden. The equals method in the Object class returns true only if the same object is both the invoking object and the parameter object. In other words, the equals method in the Object class returns false when the invoking object and the parameter object are really two distinct objects no matter what values their instance variables have. Every class in Java inherits from the Object class. It is typically the case that

classes will override the equals method that is inherited from Object because it is the instance variables of the objects that will be inspected to determine if the two objects should be considered as equal.

The compareTo method in the String class is used to compare the invoking String to the parameter String and return an int result that can be interpreted in the context of alphabetical ordering. In general, use the following format for an invocation of (call to) the compareTo method:

```
<invoking String>.compareTo (<argument String>)
```

Internally, letters of the alphabet are coded with 'a' less than 'b', 'b' less than 'c', … 'y' less than 'z'. Similarly, 'A' is less than 'B'; 'B' is less than 'C', … 'Y' is less than 'Z'. Another fact you need to know is that the entire upper case alphabet is encoded into binary values smaller than 'a'. Hence, "Zebra" is less than "aardvark" because 'Z' is less than 'a'.

The compareTo method returns a negative int if the <invoking String> comes before the <argument String> according to the internal Unicode representations of the Strings. For example

```
("cat").compareTo("dog")
```

returns a negative int value.

The compareTo method returns a positive int if the <invoking String> comes after the <argument String> according to the internal Unicode representations of the Strings.

The compareTo method returns zero if the <invoking String> is identical to the <argument String> according to the internal Unicode representations of the Strings.

To use compareTo in a conditional expression, you must use a relational operator to compare the int returned by compareTo to zero. For example, the following is a boolean valued expression

```
("cat").compareTo("dog") < 0
```

that evaluates to true. In general, you might think of

```
("cat").compareTo("dog") <operator> 0
```

as

```
("cat")<operator> ("dog")
```

The compareTo and the equals methods of the String class are consistent in that s1.equals(s2) is true if and only if s1.compareTo(s2) == 0 is true.

The String class has a compareTo method that works as described above because the String class implements the Comparable<String> interface. A Java interface provides a list of methods (signature and return type) whose details must be supplied by any class that implements the interface.

The Comparable<T> interface looks like the following:

```
public interface Comparable<T> {
        int compareTo(T obj);
}//Comparable<T>
```

Hence, any class that implements the Comparable<T> interface must provide the details of a method named compareTo that has one parameter of class T and returns an int value. The <T> after Comparable is a *type parameter*. We will discuss type parameters in the next chapter. Each specific class that implements Comparable<T> will supply a type for T. In the case of the String class, it makes sense that the type parameter T will be replaced by String. This is reflected in that the header for the String class includes "implements Comparable<String>".

Writing an equals Method for the Name Class[1]

Using the equals method of the String class as a model, let us write an equals method for our Name class. This method will be invoked by a Name object and will return true when that invoking object has all instance variables equal to the corresponding instance variables of the argument supplied to the method. A typical invocation of the equals method will have the following format:

```
<invoking Name object>.equals (<argument Name object>)
```

We use the signature and return type of the equals method of the String class as a model for the equals method of the Name class. The return type of the equals method is boolean and the equals method will have one parameter of class Object. Hence, the equals method of the Name class will override the inherited equals method of the Object class. Note in the code below that myFirst, myMiddle, and myFamily when occurring alone refer to the invoking object. We need to produce a Name reference for the parameter in order to reference its instance variables. We cannot simply use an Object reference and expect that the compiler to agree that an Object reference has instance variables named myFamily, myFirst, and myMiddle. We declare a reference to a Name object, theName, and assign it the result of casting the parameter to class Name. If somehow the parameter sent to the equals method is not really a reference to a Name object, the code below will throw a ClassCastException to indicate the problem.

```
public boolean equals(Object theOther) {
        Name theName = (Name)theOther;
        return myFamily.equals(theName.myFamily)
        && myFirst.equals(theName.myFirst)
        && myMiddle.equals(theName.myMiddle);
}// equals
```

Writing a compareTo Method for the Name Class

Let's do the work to make our Name class implement the Comparable<Name> interface. First we add to the class header to declare our intention to implement Comparable<Name> as follows:

```
public class Name implements Comparable<Name> {
```

[1]Whenever we override the equals method of the Object class and we want to use some of the data structures supplied by the Java API, we also must override the hashCode method of the Object class.

Next, we provide a compareTo method that has one Name parameter and returns an int value that is negative if the invoking Name is before the parameter Name, zero if the invoking Name equals the parameter Name, and positive if the invoking Name is after the parameter Name. We use the int values returned by the String class's compareTo method applied to each corresponding instance variable of the invoking Name and the parameter Name. We begin by comparing the myFamily instance variables. If String's compareTo returns a value other than zero, we can simply return that value and be done. If String's compareTo returns zero as the result of comparing the myFamily instance variables, we must go on to compare the myFirst instance variables. If String's compareTo returns a value other than zero for the comparison of the myFirst instance variables, we can simply return that value and be done. If String's compareTo returns zero for the myFirst instance variables, we must proceed to compare the myMiddle instance variables and can return that result provided by String's compareTo. The code is as follows:

```
public int compareTo(Name theName) {
    int compareFamily = myFamily.compareTo(theName.myFamily);
    if (compareFamily != 0) {
           return compareFamily;
    }// family names were different
    int compareFirst = myFirst.compareTo(theName.myFirst);
    if (compareFirst != 0) {
        return compareFirst;
    }// first names were different with identical family names
    return myMiddle.compareTo(theName.myMiddle);
}// compareTo
```

Note that since the code in the compareTo method selects among return statements, we do not need to use else.

Exercise 11.2: Write a before method for the Name class. The before method should have one Name parameter and have boolean return type. The before method needs only one line of code. The before method should return true only if the compareTo method returns a negative value when passed the same parameter.

Exercise 11.3: Write an after method for the Name class. The after method should have one Name parameter and have boolean return type. The after method needs only one line of code. The after method should return true only if the compareTo method returns a positive value when passed the same parameter.

Exercise 11.4: Write a `toString` method for the `Name` class. The `toString` method should return the family name, followed by a comma and a space, followed by the first name, followed by a space, followed by the middle name. For example, the `toString` method could return:

```
Mann, Lydia Michelle
```

If a middle name has not been specified and the `Name` constructor has assigned the empty string `""` to the `myMiddle` instance variable, your `toString` method should work appropriately. In other words your `toString` method could return:

```
Smith, Fred
```

The `NameList` Class

Instance Variables of the `NameList` Class

As mentioned in the introduction to this chapter, the `NameList` class will have two instance variables: an array of `Name` objects and an `int` that holds the count of `Name` objects currently in the list. Hence, the `NameList` class will begin as follows:

```
public class NameList {
    private Name myArray[]; //refers to the array of Names
    private int myCount; // holds the current count of Names in the list
```

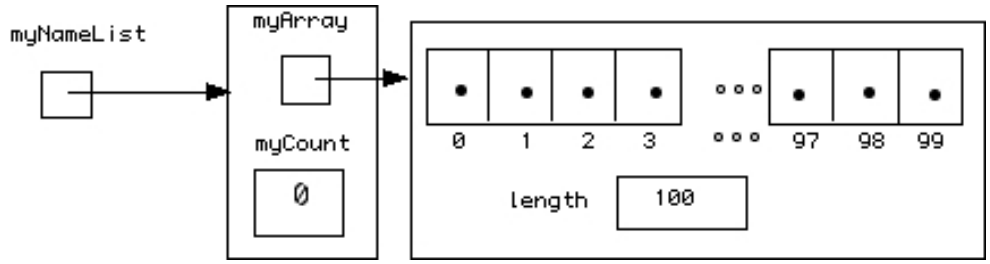
Constructors for the `NameList` Class

We will write two constructors for our `NameList` class. One constructor will have no parameters and use a default value of `100` for the length of the array `myArray`. The second constructor will have one parameter that will be used as the length of the array `myArray` when it is constructed. In both constructors, we initialize `myCount` to zero to indicate that our `NameList` objects are empty just after they are constructed. Below are the zero parameter and one parameter constructors.

```
public NameList () {
        myArray = new Name[100];
        myCount = 0;
}// 0 parameter constructor

public NameList (int theMax) {
        myArray = new Name[theMax];
        myCount = 0;
}// 1 parameter constructor
```

If an application were to execute the following instruction on a reference to a `NameList` object, the situation in memory that results is shown below.

```
myNameList = new NameList();
```



Instance Methods for the NameList Class

We will write several methods for our NameList class.

1) In order for an object of the NameList class to be useful, we will need to be able to add Name objects to the list. Hence, our NameList class should have an *add* method that has a Name object as a parameter. Our add method will have the return type boolean. The *add* method should add the parameter to the end of the list. The *add* method should return true if the Name object was successfully added to the list or return false if the Name object was not added to the list.
2) Since the list is implemented as an array, it can contain a fixed maximum number of Name objects. Hence, it will be useful for us to have a boolean valued method named isFull that returns true if the array is full, meaning that no more names can be added to the list.
3) We will also want to be able to search for a specific name in the list. Hence, we will write a search method whose return type is Name. If the search method locates a Name object in the list that is equal to a supplied parameter, the search method will return a reference to that Name object. If an equal Name object is not found, the search method will return a null reference.
4) Finally, we will want to sort the list into alphabetical order and hence, we will write a sort method for our NameList class.

The *add* Method

The code written in the *add* method should be defensive. It should check to make sure that the array is not full before adding the parameter to the list. Our code below uses the length instance variable that accompanies any array reference to accomplish this test. In addition, we use the post-increment operator to increment the myCount instance variable and save a line of code.

```
public boolean add(Name theName) {
    if (myCount < myArray.length ) {
        myArray[myCount++] = theName;
        return true;
    } // add was successful
    System.out.println("Attempting to add to full array");
    return false;
}//add
```

Note that if the `return true;` statement is executed, no other code in the method is executed. Hence, an `else` clause could be used, but is not really necessary and would violate Java coding conventions.

The isFull Method

The `isFull` method should simply return `true` if the `myCount` instance variable has the same value as the `length` of the array and `false` otherwise. This method can be written with an `if` statement, but we have chosen to write the most concise code by returning the appropriate `boolean` valued expression.

```
public boolean isFull() {
        return myCount == myArray.length;
}//isFull
```

Note that we could have written this method with an `if` statement as follows:

```
public boolean isFull() {
        if( myCount == myArray.length) {
            return true;
        }//if
        return false;
}//isFull
```

Also note that we could have used the `isFull` method in our `add` method as follows: Pay particular attention to the use of the `!` for logical negation.

```
public boolean add(Name theName) {
        if (!isFull()) {
                myArray[myCount++] = theName;
                return true;
        } // add was successful
        System.out.println("Attempting to add to full array");
        return false;
}//add
```

A toString Method for the NameList Class

We have made good use of `for` loops in conjunction with arrays. In fact, we can think of the following first line of a `for` loop as the template for processing each element in an array.

```
for (int i = 0; i < <actual number of elements in the array> ; i++) {
    //process array element with index i

}//for
```

We can put this template to use in producing a `String` representation of a `NameList` object. In this case we will be overriding the `toString` method that `NameList` inherits from the `Object` class. We will start with an empty `String` and concatenate in turn each

element of the list, relying on the `toString` method supplied by the `Name` class. The code for our method is as follows:

```
public String toString() {
    String stringToReturn = "";
    for (int i = 0; i < myCount; i++) {
        stringToReturn = stringToReturn +
            myArray [i].toString()+"\n"; //"\n" for the new line character
    }//for
    return stringToReturn;
}//toString
```

We can shorten the code by using the `+=` operator as follows:

```
stringToReturn += myArray [i].toString()+"\n";
```

We can further shorten the code by remembering that the `+` represents string concatenation when one of its operands is a `String`. In that case, the type of the other operand will be coerced to `String`. Hence, it will be done implicitly for us and we do not need to invoke the `toString` method of class `Name` explicitly. Our further shortened code is as follows:

```
stringToReturn += myArray [i]+"\n";
```

Hence, our final version of the `toString` method of class `NameList` is:

```
public String toString() {
    String stringToReturn = "";
    for (int i = 0; i < myCount; i++) {
        stringToReturn += myArray [i]+"\n"; //"\n" for the new line character
    }//for
    return stringToReturn;
}//toString
```

The search Method and the while Loop

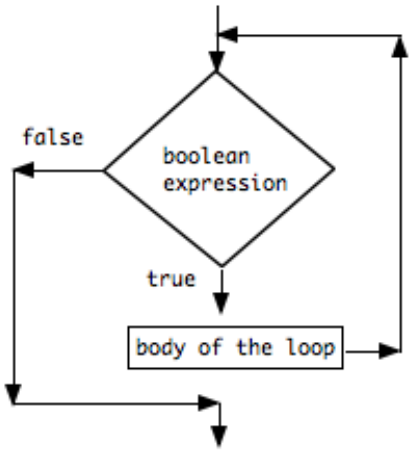
Now, we want to search for an item in an array, and if we find it, return the reference to it. In this case, we want to stop processing elements in the array if we find the element for which we are looking. A `for` loop is not really appropriate here. We need a Java statement that supports indefinite iteration. The `while` statement does the job for us. Recall from Chapter 6, the syntax of the `while` statement is as follows:

```
while ( <boolean valued expression> ) {
    body of the loop
}// while
```

The semantics of the `while` statement are as follows.

1) The `boolean` valued expression is evaluated.
2) If the expression is `true`, the body of the loop is executed and flow of execution loops back to the top of the loop where the `boolean` valued expression is found (i.e., back to step 1), and otherwise, the body of the loop is skipped and execution proceeds to the statement after the body of the loop.

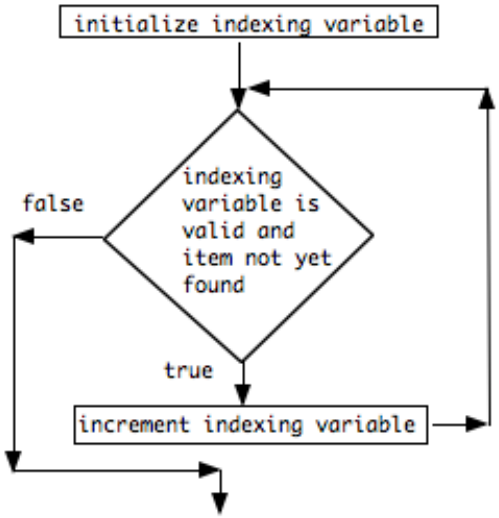
The semantics of the `while` loop are described visually in the following flow diagram.



In the case of searching for a particular item in a list, we must declare and initialize an indexing variable for the array before we evaluate the `boolean` expression. In addition, the `boolean` expression must function in two ways. In general, the `boolean` expression should evaluate to `true` if we want the loop body executed again. As it did in the case of the `for` loop, the boolean expression checks to make sure that the indexing variable `i` is still valid. In general, `i` should be less than the value of `myCount` in order to be valid. In addition, when we are searching, we want to continue searching as long as we have not already found the item for which we are searching. Hence, the condition for the `while` loop to continue iteration can be expressed as :

```
<indexing variable is valid> AND
    <current array element is not equal the item for which we are searching>
```

It is important that we check the validity of the index *before* we use it to access an array element. The body of the loop needs only get us ready to test the next element of the array. Hence, it simply increments the indexing variable `i`. The searching loop is displayed visually below.



Consider the possible situations that could exist upon exit from the `while` loop:

1) It may be that the item was not found in the list. In this case the loop ceases iteration because the indexing variable `i` is equal to `myCount` and is invalid.
2) It may be that the item was found in the list. In this case, the indexing variable `i` should still be less than `myCount`.

Because there are two possible situations, a selection statement is needed to determine the value to be returned, a reference to the found item or a `null` reference.

The following `search` method conforms to the discussion above.

```
public Name search (Name theName) {
        int i = 0;
        while ( i < myCount && !(myArray[i].equals(theName))){
            i++;
        }//while
        return (i < myCount ? myArray[i] : null);
}//search
```

sort Methods

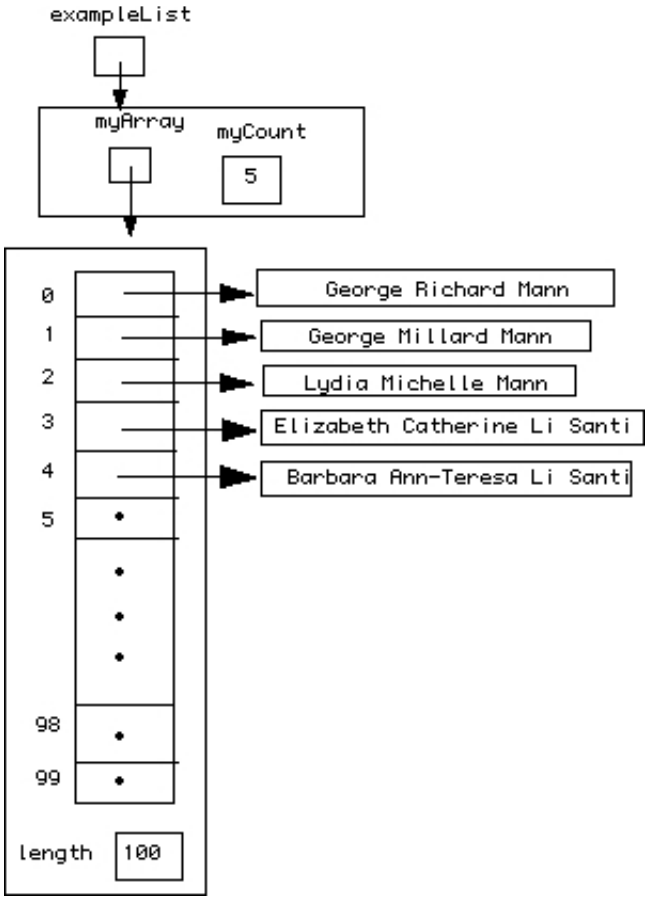
There are many different ways to sort arrays. We will discuss three different methods. We will write the code for one method and leave the coding of the other two methods as exercises.

Bubble Sort

In bubble sort, adjacent array elements are compared and if found to be out of order with respect to each other, the references are swapped to put the two compared elements in order. Many pairs of array elements must be compared to accomplish complete sorting. Let us look at a short list of 5 `Name` objects and see how bubble sort would get them into alphabetical order. Recall that our `add` method has `boolean` return type. We can use that fact to have the code test for each successful addition to the list. We will construct a `NameList` object and add 5 `Name` objects to the list as follows:

```
NameList exampleList = new NameList();
if (exampleList.add(new Name("George", "Richard", "Mann")) &&
    exampleList.add(new Name("George", "Millard", "Mann")) &&
    exampleList.add(new Name("Lydia", "Michelle", "Mann")) &&
    exampleList.add(new Name("Elizabeth", "Catherine", "Li Santi")) &&
    exampleList.add(new Name("Barbara", "Ann-Teresa", "Li Santi")) ) {
    System.out.println("Five names added to the list.");
} else {
    System.out.println("Trouble in adding 5 names to the list.");
} // else
```

Assuming that all 5 adds were successful, in memory we have the following `NameList` object named `exampleList`:



When we compare array elements in indices `0` and `1`, we find that they are out of order because George Millard Mann comes before George Richard Mann. Hence, we swap them and get the following list:

0. George Millard Mann
1. George Richard Mann
2. Lydia Michelle Mann
3. Elizabeth Catherine Li Santi
4. Barbara Ann-Teresa Li Santi

We continue to compare by comparing array elements in indices `1` and `2`. We find that they are in order since Lydia Michelle Mann is after George Richard Mann. We continue to compare by comparing array elements in indices `2` and `3`. We find again that they are out of order since Elizabeth Catherine Li Santi is before Lydia Michelle Mann. Hence, we swap the references and have the following list.

0. George Millard Mann
1. George Richard Mann
2. Elizabeth Catherine Li Santi
3. Lydia Michelle Mann
4. Barbara Ann-Teresa Li Santi

We continue to compare by comparing array elements in indices 3 and 4. We find again that they are out of order since Barbara Ann-Teresa Li Santi is before Lydia Michelle Mann. Hence, we swap the references and have the following list.

0. George Millard Mann
1. George Richard Mann
2. Elizabeth Catherine Li Santi
3. Barbara Ann-Teresa Li Santi
4. Lydia Michelle Mann

Note now that we have completed one pass through the list and we have accomplished getting the name that is last in alphabetical order into the last position. That name does not need to be involved in subsequent comparisons. It is as though we now have a list of four names to alphabetize and those names are in positions 0 through 3.

We begin the second pass over the array by comparing array elements in indices 0 and 1. We find that they are in order. We continue by comparing array elements in indices 1 and 2. We find that they are out of order since Elizabeth Catherine Li Santi is before George Richard Mann. Hence, we swap the references and have the following list.

0. George Millard Mann
1. Elizabeth Catherine Li Santi
2. George Richard Mann
3. Barbara Ann-Teresa Li Santi
4. Lydia Michelle Mann

We continue by comparing array elements in indices 2 and 3. We find again that they are out of order since Barbara Ann-Teresa Li Santi is before George Richard Mann. Hence, we swap the references and have the following list:

0. George Millard Mann
1. Elizabeth Catherine Li Santi
2. Barbara Ann-Teresa Li Santi
3. George Richard Mann
4. Lydia Michelle Mann

We have finished the second pass over the list and the two names that are last in alphabetical order are now last in the list. We now have a list of three names to sort and those names are in index positions 0 through 2.

We begin the third pass over the list by comparing array elements in indices 0 and 1. We find that they are out of order since Elizabeth Catherine Li Santi is before George Millard Mann. Hence, we swap the references and have the following list.

0. Elizabeth Catherine Li Santi
1. George Millard Mann
2. Barbara Ann-Teresa Li Santi
3. George Richard Mann
4. Lydia Michelle Mann

We continue to compare by comparing array elements in indices 1 and 2. We find that they are out of order since Barbara Ann-Teresa Li Santi is before George Millard Mann. Hence, we swap the references and have the following list.

0. Elizabeth Catherine Li Santi
1. Barbara Ann-Teresa Li Santi
2. George Millard Mann
3. George Richard Mann
4. Lydia Michelle Mann

We have finished the third pass over the list and the three names that are last in alphabetical order are now last in the list. We now have a list of two names to sort and those names are in index positions 0 and 1.

We begin the fourth and final pass over the list by comparing array elements in indices 0 and 1. We find that they are out of order since Barbara Ann-Teresa Li Santi is before Elizabeth Catherine Li Santi. Hence, we swap the references and have the following list which is completely alphabetized since there is only a list of one element left to be alphabetized and that list is automatically in order.

0. Barbara Ann-Teresa Li Santi
1. Elizabeth Catherine Li Santi
2. Lydia Michelle Mann
3. George Millard Mann
4. George Richard Mann

We need to analyze the pattern of comparisons and abstract that analysis to any size list. The first pass over the list included the following comparisons denoted here as ordered pairs: (0,1), (1,2), (2,3), (3,4). The second pass made the following position comparisons: (0,1), (1,2), (2,3). The third pass made the following position comparisons: (0,1), (1,2). Finally, the last pass made only one comparison (0,1). In each of these comparisons, the second position is one more than the first position. Hence we could abstractly represent all of our ordered pairs as (j, j+1). In each pass j starts at 0. But in each pass the final value of j is one less than it was in the previous pass. Let us look further for a pattern.

pass number	final value of j
1	3
2	2
3	1
4	0

Remember that there were 5 names in the list because myCount had value 5. If we had an int variable to count the passes over the list named passNum, then the final value of j in each pass would be

```
myCount – passNum – 1
```

From a top-down design perspective, we need a `for` loop that counts the pass numbers from `1` through `myCount – 1`:

```
for(int passNum = 1; passNum < myCount; passNum++) {
    //code for one pass here
}//for loop that controls passes over the data
```

Within that `for` loop, we need another `for` loop with loop control variable `j`. Each time the inner loop is executed `j` begins at `0` and ends at `myCount – passNum – 1`.

```
for(int passNum = 1; passNum < myCount; passNum++) {
    for (int j = 0; j < myCount – passNum; j++) {
        // put comparison and possible swap code here for (j, j+1)
    }//inner for loop for one pass
}//for loop that controls passes over the data
```

The comparison and swap code is dependent on a `before` method in the `Name` class (Exercise 11.2). We want to do a swap of references if the element in position `j+1` is `before` the element in position `j`. We can carefully write the correct selection statement as follows:

```
if(myArray[j+1].before(myArray[j]) {
        swap(j, j+1);
}//if
```

Note the use of top–down design once more. We put off the details of the swapping by invoking a `swap` method that takes the two `int` positions as arguments. The `swap` method can be `private` in the `NameList` class and is similar to the `swap` method you saw in the `Deck` class in Chapter 10.

```
private void swap (int firstPos, int otherPos) {
    Name temp = myArray[firstPos];
    myArray[firstPos] = myArray[otherPos];
    myArray[otherPos] = temp;
}//swap
```

We have assembled all of the code for the `bubbleSort` method below.

```
public void bubbleSort() {
    for(int passNum = 1; passNum < myCount; passNum++) {
        for (int j = 0; j < myCount – passNum; j++) {
            if(myArray[j+1].before(myArray[j]) {
                swap(j, j+1);
            }//if
        }//inner for loop for one pass
    }//for loop that controls passes over the data
}//bubbleSort
```

Selection Sort

The Selection Sort algorithm is more efficient than Bubble Sort. In Bubble Sort whenever two array elements are found to be out of order, they are swapped. The swap requires the allocation of a temporary variable and three assignment statements. The Selection Sort algorithm greatly reduces the number of swaps performed. The strategy for Selection Sort is as follows:

1) Find the position of the element that is the smallest value stored in the array. Store that position in an `int` variable named `smallest`.
2) If the value of `smallest` is not `0` (meaning that the item in position `0` is not the smallest in the array), swap the references at positions `0` and `smallest`.
3) The smallest item is now in position `0`. Only `myCount - 1` items need to be sorted and they are in positions `1` through `myCount - 1`. Find the position of the smallest element in this shorter list and store that position in `smallest`.
4) If the value of `smallest` is not `1` (meaning that the item in position `1` is not the smallest in the remainder of the list), swap the references at positions `1` and `smallest`.
5) The two smallest items are now appropriately in positions `0` and `1` of the array. Only `myCount - 2` items need to be sorted and they are in positions `2` through `myCount - 1`. Find the position of the smallest element in this shorter list and store that position in `smallest`.
6) If the value of `smallest` is not `2` (meaning that the item in position `2` is not the smallest in the remainder of the list), swap the references to positions `2` and `smallest`.

You can see that this process will be repeated on smaller and smaller numbers of elements in the list. The last time we need to find a smallest element, we are looking at a two item list occupying positions `myCount - 2` and `myCount - 1` in the array. In addition, it should be clear that far fewer swaps are performed. In fact there will be at most one swap for each pass over the list or `myCount - 1` swaps. In Bubble Sort the maximum number of swaps is one per comparison. There were 4 + 3 + 2 + 1 or 10 swaps possible on a list of 5 elements. Selection Sort would have a maximum of 4 swaps. The comparison of efficiency is even more obvious on an array of 100 elements. Selection Sort will perform at most 99 swaps. Bubble Sort will perform at most 99 + 98 + 97 + 96 + … + 3 + 2 + 1 or 4950 swaps. This could be called a worst case assessment of the situation.

Returning to the Selection Sort algorithm, the question now focuses on how to find the smallest item in an list. We begin by guessing that the smallest item is in the first position we have under consideration and assign that guess to the `int` variable named `posOfSmallest`. We then use the `before` method to compare in turn each succeeding element in the list with the element in the array in position indicated by the value of `posOfSmallest`. We replace the value of `posOfSmallest` whenever we find a better candidate for the smallest element, i.e., whenever the next item we consider is smaller than the one we currently think is the smallest.

For example, consider the list we used above to examine Bubble Sort.

0. George Richard Mann
1. George Millard Mann
2. Lydia Michelle Mann
3. Elizabeth Catherine Li Santi
4. Barbara Ann-Teresa Li Santi

We begin by finding the position of the smallest element in the range of positions `0` through `4`. We guess that it is in position `0` to start (George Richard Mann). Hence, the `int` variable `posOfSmallest` will be initialized with value `0`. We then compare the element in position `1` with the element in position `0` and determine that the item in position `1` is a better candidate. We assign the value `1` to the `int` variable `posOfSmallest`. Our next comparison is between the element we currently think is smallest in position `1` and the element in position `2`. The element in position `2` is not a better candidate for the smallest. Next we compare the value in position `3` to the value in the position indicated by `posOfSmallest` (`1`) and find that the element in position `3` is a better candidate for the smallest. We assign the value `3` to `posOfSmallest`. Finally we compare the element in position `4` with the element in the position indicated by `posOfSmallest` (`3`) and find that the element in position `4` is a better candidate for smallest. We assign the value `4` to `posOfSmallest` to record that finding. At this point we have determined that the element in position `4` is the smallest in the array and make the swap between the elements in positions `0` and `4` to get the following list.

0. Barbara Ann-Teresa Li Santi
1. George Millard Mann
2. Lydia Michelle Mann
3. Elizabeth Catherine Li Santi
4. George Richard Mann

Note that we need now only consider array elements in positions `1` through `4`. We need to find the position of the smallest element of that section of the array. We start by guessing that the element that belongs in position `1` is already there, i.e., that the smallest of this section is in position `1`. We perform the comparisons as described above and end up with the value `3` for `posOfSmallest`. We then swap the elements in positions `1` and `3` and arrive at the following list:

0. Barbara Ann-Teresa Li Santi
1. Elizabeth Catherine Li Santi
2. Lydia Michelle Mann
3. George Millard Mann
4. George Richard Mann

Now we consider elements in positions `2` through `4`. Our goal is to find the smallest of these elements. We start by guessing that the smallest is in position `2`. After performing comparisons, we find that the smallest is in position `3` and swap the elements in positions `2` and `3`. The list now becomes:

0. Barbara Ann-Teresa Li Santi
1. Elizabeth Catherine Li Santi
3. George Millard Mann

3. Lydia Michelle Mann
4. George Richard Mann

We now consider the elements in positions 3 and 4. We find the smallest in position 4 and perform the swap between positions 3 and 4 to get the following list:

0. Barbara Ann-Teresa Li Santi
1. Elizabeth Catherine Li Santi
3. George Millard Mann
3. George Richard Mann
4. Lydia Michelle Mann

Since all but the last array element have been put into order, the last element falls automatically in order and the list is sorted.

Let us try to analyze again in general. For each of the positions 0 through myCount – 2, we need to find the appropriate element for that position. For position 0, we search for the smallest item in positions 0 through myCount – 1. For position 1, we search for the smallest item in positions 1 through myCount – 1. For position 2, we search for the smallest item in positions 2 through myCount – 1. It appears that the search area for position j is positions j through myCount – 1.

We could express the algorithm in pseudocode as follows:

For each position named fixPos from position 0 through position myCount – 2, do the following:

1) Find the position of the smallest array element in positions fixPos through myCount – 1 and store that index value in a variable named posOfSmallest.
2) If the values of fixPos and posOfSmallest are different, swap the references at those positions.

Exercise 11.5: Write the code for method selectionSort of the NameList class using the discussion above.

Insertion Sort

A third sort algorithm simulates a process that people often go through when playing cards and arranging the cards in a hand. The first card is in order when it is alone. After a second card is dealt, it is placed in the hand in appropriate position with respect to the first card so that the two cards are then in order. After a third card is dealt, it is placed in the hand in appropriate position with respect to the first two cards so that the three cards are then in order. In general, when the nth card is dealt, it is placed in the appropriate place with respect to the first n–1 cards that are already in order. This process of inserting the new card in appropriate position with respect to the previous cards is the main strategy used in Insertion Sort. Once more consider our list of names:

0. George Richard Mann
1. George Millard Mann
2. Lydia Michelle Mann

3. Elizabeth Catherine Li Santi
4. Barbara Ann-Teresa Li Santi

At first consider the list as having just one name:

0. George Richard Mann

This list is in order. Now consider the second name in the list to be inserted with respect to the first name in the list. Since George Millard Mann is before George Richard Mann, we need to move George Richard Mann down in the list to position `1` because George Millard Mann really belongs in position `0` in our two item list. Hence, we rearrange the list as follows:

0. George Millard Mann
1. George Richard Mann

Now consider that we need to insert Lydia Michelle Mann into our list of two names. Since Lydia is after both of the names already in the list, we can leave her in the position she currently occupies, i.e., position `2`.

0. George Millard Mann
1. George Richard Mann
2. Lydia Michelle Mann

The next name to consider is Elizabeth Catherine Li Santi. Since Elizabeth is before all of the names currently in the list, we effectively need to move those three names down into positions `1`, `2`, and `3` to make position `0` available for Elizabeth. We get the resulting list:

0. Elizabeth Catherine Li Santi
1. George Millard Mann
2. George Richard Mann
3. Lydia Michelle Mann

Finally, we need to insert Barbara Ann-Teresa Li Santi into the list. Since Barbara is before all of the names on the list, we need to move them down into positions `1` through `4` to make room for Barbara in position `0`. We get the following list:

0. Barbara Ann-Teresa Li Santi
1. Elizabeth Catherine Li Santi
2. George Millard Mann
3. George Richard Mann
4. Lydia Michelle Mann

The major conceptual difference between what we have described above and the general Insertion Sort is that the elements of the array are all present from the beginning. Hence the array changes as follows:

0. George Richard Mann
1. George Millard Mann

2. Lydia Michelle Mann
3. Elizabeth Catherine Li Santi
4. Barbara Ann-Teresa Li Santi

First insert the second item (George Millard Mann) in its proper place with respect to the first item:

0. George Millard Mann
1. George Richard Mann
2. Lydia Michelle Mann
3. Elizabeth Catherine Li Santi
4. Barbara Ann-Teresa Li Santi

Now insert the third item (Lydia Michelle Mann) in its proper place with respect to the first two items. The list does not change in this process.

Now insert the fourth item (Elizabeth Catherine Li Santi) in its proper place with respect to the first three items. The list becomes:

0. Elizabeth Catherine Li Santi
1. George Millard Mann
2. George Richard Mann
3. Lydia Michelle Mann
4. Barbara Ann-Teresa Li Santi

Finally insert the fifth item (Barbara Ann-Teresa Li Santi) in its proper place with respect to the first four items. The list becomes:

0. Barbara Ann-Teresa Li Santi
1. Elizabeth Catherine Li Santi
2. George Millard Mann
3. George Richard Mann
4. Lydia Michelle Mann

The strategy for Insertion Sort can be described in the following pseudocode:

For each position named `toBeInserted` from position `1` through position `myCount - 1`, do the following:

1) Find the appropriate position for the element in position `toBeInserted` among the elements in positions `0` through `toBeInserted`. Assign that position to an `int` variable named `correctPos`.
2) If the values of `toBeInserted` and `correctPos` are different, do the following
    a) Store a reference to the object in the array in position `toBeInserted` in a temporary variable named `temp`.
    b) Move elements in the array in positions `toBeInserted - 1` through `correctPos` down one position in the array.
    c) Assign the reference `temp` to the array in position `correctPos`.

Exercise 11.6: Write an `insertionSort` method for the `NameList` class based on the discussion above.

Exercise 11.7: Assuming that the array in a `NameList` object has been sorted, the search algorithm can be made more efficient. An item can be determined *not* to be in the list as soon as a list element is detected that is after the item for which we are searching. Revise the `search` method to create a `searchSorted` method for the `NameList` class that takes advantage of the list being sorted when it searches for a particular `Name` object.

Binary Search Algorithm

Even though a phone directory for a college or university is a sorted list, we would never search for the name Carol Lennox by starting with the first name in the list and checking each name in order until we either find Carol Lennox or find a name after Carol Lennox to indicate that Carol Lennox is not in the list. Instead, we might check a name at approximately the middle of the list, and if it is not Carol Lennox, it is a name that is either before or after Carol Lennox. If that name is after Carol Lennox, so is every other name beyond it in the list. We can then eliminate the entire second half of the list when we continue to search for Carol Lennox. We can perform this process repeatedly halving the list until the name is found in the middle, or there is no section of the list remaining to search. Let us consider this algorithm known as Binary Search on the following list of 20 names.

0. Linda Adams
1. Elizabeth Baggan
2. Susan Bass
3. Mildred Campbell
4. Evalyn Clark
5. Dennis Handler
6. Kinley Kraft
7. Carol Lennox
8. Barbara Li Santi
9. Jan MacIntosh
10. Lydia Mann
11. Diane McEntyre
12. Marilyn Mendle
13. Eileen Nadler
14. Michele Proust
15. Diane Resek
16. Nancy Robertson
17. Amy Sandman
18. Elizabeth Stone
19. Fred Zenith

We have the entire list to search at the start of the algorithm. This can be indicated by storing a value for the lowest index currently under consideration and the highest index under consideration. In this case low is 0 and high is 19. In order to determine a middle index, we use the mathematical concept of average and calculate the value of (low + high)/2. In this case we get 19/2 which is 9.5 if we do arithmetic with floating point numbers. But if we do integer division, 19/2 is 9. Hence, we will use 9 as our middle index. The name in position 9 is Jan MacIntosh. Since Jan MacIntosh is alphabetically after Carol Lennox, we can eliminate Jan MacIntosh and all of those names after her from consideration. We accomplish that elimination of part of the list by assigning the value 8 to high. Note that 8 is one less than the current value of our middle index. Hence, the portion of our list to search has been reduced to:

0. Linda Adams
1. Elizabeth Baggan
2. Susan Bass
3. Mildred Campbell
4. Evalyn Clark
5. Dennis Handler
6. Kinley Kraft
7. Carol Lennox
8. Barbara Li Santi

We calculate a new value for middle. (low + high)/2 is (0 + 8)/2 or 4. The name in position 4 is Evalyn Clark. Since Carol Lennox is after Evalyn Clark in alphabetical order, we can discard Evalyn Clark and all names before her. We do that by changing the value of low to 5 which is one more than the current value of the middle position. Hence, the portion of our list to search has been reduced to:

5. Dennis Handler
6. Kinley Kraft
7. Carol Lennox
8. Barbara Li Santi

We calculate a new value for middle. (low + high)/2 is (5 + 8)/2 or 13/2 or 6. The name in position 6 is Kinley Kraft. Since Carol Lennox is after Kinley Kraft, we can discard Kinley Kraft and any names before her from the search area. We do that by changing the value of low to become 7 or one more than the current value of the middle position. Hence, the portion of our list to search has been reduced to:

7. Carol Lennox
8. Barbara Li Santi

We calculate a new value for middle. (low + high)/2 is (7 + 8)/2 or 15/2 or 7. The name in position 7 is Carol Lennox. We have found the name for which we were searching!!

Now let us see what happens if we search for a name that is not in the list, Lisa Lemon.

Begin again with the entire list:

0. Linda Adams
1. Elizabeth Baggan
2. Susan Bass
3. Mildred Campbell
4. Evalyn Clark
5. Dennis Handler
6. Kinley Kraft
7. Carol Lennox
8. Barbara Li Santi
9. Jan MacIntosh
10. Lydia Mann
11. Diane McEntyre
12. Marilyn Mendle
13. Eileen Nadler
14. Michele Proust
15. Diane Resek
16. Nancy Robertson
17. Amy Sandman
18. Elizabeth Stone
19. Fred Zenith

In this case low is 0 and high is 19. We calculate the value of (low + high)/2 which is (0 + 19)/2 or 9. The name in position 9 is Jan MacIntosh. Since Jan MacIntosh is alphabetically after Lisa Lemon, we can eliminate Jan MacIntosh and all of those names after her from consideration by assigning the value that is one less than the current value of our middle index (8) to high. Hence, the portion of our list to search has been reduced to:

0. Linda Adams
1. Elizabeth Baggan
2. Susan Bass
3. Mildred Campbell
4. Evalyn Clark
5. Dennis Handler
6. Kinley Kraft
7. Carol Lennox
8. Barbara Li Santi

We calculate a new value for middle. (low + high)/2 is (0 + 8)/2 or 4. The name in position 4 is Evalyn Clark. Since Lisa Lemon is after Evalyn Clark in alphabetical order, we can discard Evalyn Clark and all names before her by changing the value of low to 5 or one more than the current value of the middle position. Hence, the portion of our list to search has been reduced to:

5. Dennis Handler
6. Kinley Kraft
7. Carol Lennox
8. Barbara Li Santi

We calculate a new value for middle. (low + high)/2 is (5 + 8)/2 or 13/2 or 6. The name in position 6 is Kinley Kraft. Since Lisa Lemon is after Kinley Kraft, we can discard Kinley Kraft and any names before her from the search area by changing the value of low to become 7 or one more than the current value of the middle position. Hence, the portion of our list to search has been reduced to:

7. Carol Lennox
8. Barbara Li Santi

We calculate a new value for middle. (low + high)/2 is (7 + 8)/2 or 15/2 or 7. The name in position 7 is Carol Lennox. Carol Lennox is after Lisa Lemon in alphabetical order. Hence, we should discard Carol Lennox and any names after her from the search area by changing high to become one less than the current value of our middle index or 6. Note now that low is 7 and high 6. This condition indicates that there is no longer a search area in the list and the name is *not* found.

Let us write the Java code for a `binarySearch` method that returns a reference to the `Name` object if it is found or returns `null` if the `Name` object is not in the list.

```
public Name binarySearch(Name nameToFind) {
    int low = 0;
    int high = myCount-1;
```

```
    int middle = (low+high)/2;
    while (low <= high && ! (myArray[middle].equals(nameToFind)) ) {
        if ( nameToFind.before (myArray[middle]) {
            high = middle-1;
        } else {
            low = middle+1;
        }//else
        middle = (low+high)/2;
    }//while
    return (myArray[middle].equals(nameToFind) ? myArray[middle] : null);
}//binarySearch
```

Exercise 11.8: Trace the binary search algorithm on the list of 20 names above in each of the following situations:

a) Search for Linda Adams.

b) Search for Diane McEntyre.

c) Search for Fred Zlotnick.

d) Search for George Mann.

Exercise 11.9:

1) Write a class named `Address` using the specifications below.
2) Write a class named `AddressInfo` using the specifications below.
3) Write a class named `AddressInfoList` using the specifications below.
4) Write a class named `TestAddressInfoList` that tests all of the code you wrote for parts 1), 2), and 3).

Specifications for the Class Address:

The class `Address` will implement the `Comparable<Address>` interface. Objects of class `Address` will have five `String` instance variables to represent the street address, an optional second line, the city, the state, and the zipcode. The class `Address` will have a **5**-parameter constructor, a **4**-parameter constructor, and a **0**-parameter constructor:

a) The first constructor takes 5 `String` parameters and assigns the parameters to the corresponding private instance variables.

b) The second constructor takes 4 `String` parameters representing the street address, the city, the state, and the zipcode and assigns the parameters to the corresponding private instance variables. The instance variable for the optional second line should be assigned a default value.

c) The third constructor takes no parameters and assigns default values to all five instance variables.

(continues)

Exercise 11.9, continued

The class `Address` will have **14** public methods:

a) An access method for the street address instance variable.

b) A modifier method for the street address instance variable.

c) An access method for the optional line instance variable.

d) A modifier method for the optional line instance variable.

e) An access method for the city instance variable.

f) A modifier method for the city instance variable.

g) An access method for the state instance variable.

h) A modifier method for the state instance variable.

i) An access method for the zipcode instance variable.

j) A modifier method for the zipcode instance variable.

k) A `boolean` valued method named `equals` that overrides the `equals` method of the `Object` class. The `equals` method takes an `Object` parameter that it casts to an `Address` object. The `equals` method returns `true` if and only if the five instance variables of the invoking object are equal to the five instance variables of the parameter.

Addresses will be ordered with the state as the primary key, the city as the secondary key, the zipcode as the third key, the street address as the fourth key, and the optional line as the last key.

l) An `int` valued method named `compareTo` that compares the invoking `Address` object to a parameter `Address` object. The comparison will use all five instance variables as described above for ordering addresses. The `compareTo` method should return:

- a **negative** value if the invoking object is **before** the parameter object,
- a **positive** value if the invoking object is **after** the parameter object,
- and should return **zero** only in the case that all five fields of the invoking object and the parameter object are identical.

m) a `boolean` valued method named `before`. The `before` method takes an `Address` parameter. The `before` method returns `true` if and only if the invoking object is before the parameter object with respect to the ordering of addresses as described above.

n) A `toString` method that returns a `String` representation of the information in an object of class `Address` in the following format: the `String` corresponding to the street address on a line by itself, the optional line by itself on a line (make sure that your code does not leave a blank line if the optional line is empty), the city followed by a comma and a space, followed by the state, followed by a space, and the zipcode on a line by themselves. In the case that all five instance variables have been assigned default values, the `toString` method should return an empty `String`.

(continues)

Exercise 11.9, continued

Specifications for the Class `AddressInfo`:

Each `AddressInfo` object will have *two* instance variables. The first instance variable will be a reference to an object of class `Name` as described earlier in this chapter. The second instance variable will be a reference to an object of class `Address`. The class `AddressInfo` will implement the `Comparable<AddressInfo>` interface.

The class `AddressInfo` will have a **8**-parameter constructor, a **7**-parameter constructor, and a **3**-parameter constructor:

a) A constructor that can be used when all 8 pieces of data about a person have been entered (3 for the `Name` and 5 for the `Address`). This constructor should invoke the appropriate constructors of classes `Name` and `Address`.

b) A constructor that can be used when 7 pieces of data about a person (no optional address line) have been specified. This constructor should invoke the appropriate constructors of classes `Name` and `Address`.

c) A three parameter constructor where the parameters correspond to the first, middle, and family names of a person. This constructor should invoke the appropriate constructors of classes `Name` and `Address`.

The class `AddressInfo` will have **9** `public` methods:

a) An access method for the `Name` instance variable.

b) An access method for the `Address` instance variable.

c) A modifier method for the `Name` instance variable.

d) An modifier method for the `Address` instance variable.

e) A `toString` method that can be used to display the data to a user. The `toString` method should invoke the appropriate `toString` methods in classes `Name` and `Address`.

f) A `boolean` valued method named `equals` that overrides the `equals` method of the `Object` class. The `equals` method takes an `Object` parameter that it casts to an `AddressInfo` object. The `equals` method returns `true` if and only if the `Name` of the invoking object and the `Name` of the parameter are equal. Note that this method only uses the `Name` instance variables of the invoking object and parameter.

g) An `int` valued method named `compareTo` that compares the invoking `AddressInfo` object to a parameter `AddressInfo` object. The comparison will be alphabetical depending on *only* the `Name` field. The `compareTo` method of class `AddressInfo` should invoke the `compareTo` method in class `Name` with the `Name` instance variable as invoking object and the `Name` instance variable of the parameter as the parameter for `Name`'s `compareTo` and it should return the value that the `compareTo` method in class `Name` returns to it.

h) A `boolean` valued method named `beforeAlpha`. The `beforeAlpha` method takes an `AddressInfo` parameter. The `beforeAlpha` method returns `true` if and only if the invoking object's `Name` is alphabetically before the parameter object's `Name`.

(continues)

Exercise 11.9, continued

i) A boolean valued method named beforeAddress. The beforeAddress method takes an AddressInfo parameter. The beforeAddress method returns true if and only if exactly one of the two following conditions is met: 1. The invoking object is before the parameter object with respect to the sorting scheme for addresses described above; or 2. The invoking object and the parameter object have equal addresses and the invoking object's Name is alphabetically before the parameter object's Name.

Specifications for the Class AddressInfoList:

An object of class AddressInfoList will have **three** fields. The first field will be an array of AddressInfo objects. The second field will be an int holding the number of records currently in the AddressInfoList. The third field will be a static final int that indicates the maximum capacity of the AddressInfoList.

The AddressInfoList class will have a zero parameter constructor that constructs an array of references to AddressInfo objects of default length equal to the static final int described above. The constructor also initializes the number of records instance variable to zero.

The AddressInfoList class will *at minimum* have the **7** following methods:

a) A boolean valued method named add that takes an AddressInfo parameter representing a person to insert into the list. If a record with the same name attribute as the parameter is *not* already in the database, the method returns true *after* inserting the new record. If a record with the same name as the parameter is already in the database, the method returns false and the original record remains unchanged. false is also returned if the list is full.

b) A String valued method named toString that returns a String representation of the entire array of AddressInfo objects.

c) A boolean valued method named isFull that returns true if the list is full and cannot hold another record.

d) A method named alphaSort that sorts the array into alphabetical order.

e) A method named addressSort that sorts the array into the address order described above.

f) An AddressInfo valued method named search that takes an AddressInfo parameter and returns a reference to the object in the list that matches the name attribute of the parameter or returns null if there is no record with matching name in the list.

g) A boolean valued method named delete that takes an AddressInfo parameter and attempts to delete a record from the list that matches the name attribute of the parameter. delete returns true if a record is deleted or returns false if no matching record is found to delete, causing the deletion to fail.

Exercise 11.10: This exercise has as its goal the creation of an application program that allows a user to maintain a simple file oriented database of address information.

A user of your database application should have the following commands available to manipulate the database:

- **Add**: puts information about a person and her/his address in the database. At minimum, the user must provide a first name, middle name, family name, street address, city, state, and zipcode. The program should NOT proceed without minimal information. An additional line of address information specifying a room, suite, etc. is an optional piece of data when information is being added to the database. As the programmer, you will need to determine how to code the situation where the optional line is empty so that this situation does not cause your code to fail in the context of other records where the optional line is present. All of the information taken together for a person shall be referred to as a "record". If a person with the same first, middle, and family name already exists in the database, a message should be displayed informing the user about the existing record and the insertion of the new record should fail. The program should also display a message to the user after each successful insertion of a new record.
- **Display**: outputs the entire contents of the database.
- **Alphabetical Sort and Display**: sorts the database in alphabetical order and then displays all records. Records should be sorted with the person's family name as the primary key, the first name as the secondary key, and the middle name as the last key.
- **Address Sort and Display**: sorts the database in address order. Addresses should be sorted with the state as the primary key, the city as the secondary key, the zipcode as the third key, the street address as the fourth key, and the optional line as the last key. In the case of two entries with the same address, sorting will continue with the family name, then the first name, and finally the middle name of the person.
- **Search**: allows the user to enter a person's name (family, first, and middle) to determine if the person is in the database. A search cannot proceed without the minimal information of a family name, a first name, and a middle name. If the person is found, her/his information should be displayed. If the person is not found, an appropriate message should be given to the user.
- **Delete**: allows the user to remove the information concerning a specified person from the database. The user should specify the family name, the first name, and the middle name of the person to be deleted. If the person is not found in the database, an appropriate message should be given to the user. Otherwise, once the record is found, it should be displayed, and the user should be asked to confirm the deletion request. The database application should give the user an opportunity to cancel the deletion request. If the user confirms the deletion request, the record should be deleted, and the user should be informed the deletion took place. If the user cancels the deletion request, the user should be informed that the record was not deleted.

(continues)

Exercise 11.10, continued

1) Create a GUI for the database application in a class named `Database`. Your GUI should include buttons for each of the possible commands: Add, Display, Alphabetical Sort And Display, Address Sort and Display, Search, and Delete (including confirming or canceling a delete request). It should also contain text fields and/or combo boxes for user input and a way to display output from the database. At this point the response to any button click should be to display a message that the particular button was clicked or a combo box element was selected.
2) Write enough code in your `Database` class so that the buttons for Add and Display in the GUI are functional. Make sure that you include error–checking for user input. In particular, make sure the user provides at least the minimum input required.
3) Complete the database application. This means that the Search, Alphabetical Sort And Display, Address Sort And Display, and Delete functions of the GUI should work. The methods `alphaSort`, `addressSort`, `search`, and `delete` of the `AddressInfoList` class support those functions of the GUI. Again, make sure input from the user is appropriately error–checked for the new commands being implemented. Remember to test your program thoroughly. Include a test where the user asks for a deletion and cancels the request after the record is found. Also include a test where the user fills the list, then deletes data, and then fills the list again.

Exercise 11.11:

1) Write a class named `BirthDate` using the specifications below.
2) Write a class named `BirthInfo` using the specifications below.
3) Write a class named `BirthInfoList` using the specifications below.
4) Write a class named `TestBirthInfoList` that tests all of the code you wrote for parts 1), 2), and 3).

Specifications for the Class `BirthDate`:

The class `BirthDate` will implement the `Comparable<BirthDate>` interface. Objects of class `BirthDate` will have three `int` instance variables to represent the month, day, and year of a person's birth. The class `BirthDate` will have a **3**-parameter constructor, a **2**-parameter constructor, and a **0**-parameter constructor:

a) The first constructor has 3 `int` parameters and assigns the parameters to the corresponding private instance variables.
b) The second constructor has 2 `int` parameters representing the month and day and assigns the parameters to the corresponding private instance variables. The instance variable for year should be assigned a default value selected by you.
c) The third constructor takes no parameters and assigns default values (selected by you) to all three instance variables.

(continues)

Exercise 11.11, continued

The class `BirthDate` will have **10** `public` methods:

a) An access method for the month instance variable.

b) A modifier method for the month instance variable. Note that only values in the range 1–12 are valid for the month. An attempt to modify the month instance variable so that it is not in the range 1–12 should result in *no* change in the value of the instance variable and should cause a message to be displayed through `System.out.println` about an attempt to assign an invalid month. If the value of the month instance variable is changed, this method must check to see if the value of the day instance variable is still valid. If the value of the day instance variable is invalid because it is larger than the largest possible value for the particular month (Months January, March, May, July, August, October, and December have days in range 1–31; April, June, September, and November have days in range 1–30; and February has days in range 1–29.), your code should assign the maximum possible value to the day instance variable.

c) An access method for the day instance variable.

d) A modifier method for the day instance variable. The day instance variable should be reassigned only if the new value is appropriate for the month. The maximum valid value for the day instance variable really depends on the month instance variable. Check that months January, March, May, July, August, October, and December have days in range 1–31; April, June, September, and November have days in range 1–30; and February has days in range 1–29. An attempt to modify the day instance variable so that it is not in the appropriate range should result in *no* change in the value of the instance variable and should cause a message to be displayed through `System.out.println` about an attempt to assign an invalid day in a specified month.

e) An access method for the year instance variable.

f) A modifier method for the year instance variable. (Optional Extra Credit: Do not allow the year instance variable to be assigned a value that is larger than the current year.)

g) a `boolean` valued method named `equals` that overrides the `equals` method of the `Object` class. The `equals` method takes an `Object` parameter that it casts to a `BirthDate` object. The `equals` method returns `true` if and only if the month, day, and year of the invoking object are equal to the month, day, and year of the parameter.

h) a `boolean` valued method named `before`. The `before` method has a `BirthDate` parameter. The `before` method returns `true` if and only if the invoking object is chronologically before the parameter object. The comparison will be chronological with the month as the first key, the day as the second key, and the year as the third key.

i) an `int` valued method named `compareTo` that compares the invoking `BirthDate` object to a parameter `BirthDate` object. The comparison will be chronological with the month as the first key, the day as the second key, and the year as the third key. The `compareTo` method should return:

- a **negative** value if the invoking object is **before** the parameter object,
- a **positive** value if the invoking object is **after** the parameter object,
- and should return **zero** only in the case that all three fields of the invoking object and the parameter object are identical.

(continues)

Exercise 11.11, continued

j) A `toString` method that returns a `String` representation of the information in an object of class `BirthDate` in the following format: the `String` corresponding to the month `int` instance variable, followed by a space, followed by the `int` representing the day, followed by a comma, a space, and the year only in cases where the year has not been assigned the default value. Make sure that if the year has the default value, the comma, the space and the default value for the year are not concatenated. In the case that all three instance variables have been assigned default values, the `toString` method should return an empty `String`.

Specifications for the class `BirthInfo`:

Each `BirthInfo` object will have **two** instance variables. The first instance variable will be a reference to an object of class `Name`. The second instance variable will be a reference to an object of class `BirthDate`. The class `BirthInfo` will implement the `Comparable<BirthInfo>` interface.

The class `BirthInfo` will have a **6**-parameter constructor, two **5**-parameter constructors, and a **3**-parameter constructor:

a) A constructor that can be used when all 6 pieces of data about a person have been entered. This constructor should invoke the appropriate constructors of classes `Name` and `BirthDate`.

b) A constructor that can be used when 5 pieces of data about a person (no year of birth) have been specified. This constructor should invoke the appropriate constructors of classes `Name` and `BirthDate`. It is the responsibility of the constructor of class `BirthDate` to assign a default value for the year of birth if needed.

c) A constructor that can be used when 5 pieces of data about a person (no middle name) have been specified. This constructor should invoke the appropriate constructors of classes `Name` and `BirthDate`.

d) A three parameter constructor where the parameters correspond to the first, middle, and family names of a person. This constructor should invoke the appropriate constructors of classes `Name` and `BirthDate`.

The class `BirthInfo` will have **9** `public` methods:

a) An access method for the `Name` instance variable.

b) An access method for the `BirthDate` instance variable.

c) A modifier method for the `Name` instance variable.

d) A modifier method for the `BirthDate` instance variable.

e) A `toString` method that can be used to display the data to a user. The `toString` method should invoke the appropriate `toString` methods in classes `Name` and `BirthDate`.

f) A `boolean` valued method named `equals` that overrides the `equals` method of the `Object` class. The `equals` method has an `Object` parameter that it casts to a `BirthInfo` object. The `equals` method returns `true` if and only if the `Name` of the invoking object and the `Name` of the parameter are equal. Note that this method only uses the `Name` instance variables of the invoking object and parameter.

g) A `boolean` valued method named `beforeAlpha`. The `beforeAlpha` method takes a `BirthInfo` parameter. The `beforeAlpha` method returns `true` if and only if the invoking object is alphabetically before the parameter object.

(continues)

Exercise 11.11, continued

h) A boolean valued method named beforeChron. The beforeChron method takes a BirthInfo parameter. The beforeChron method returns true if and only if exactly one of the two following conditions is met: 1. The invoking object is chronologically before the parameter object; or 2. The invoking object and the parameter object are chronologically equal and the invoking object is alphabetically before the parameter.

i) an int valued method named compareTo that compares the invoking BirthInfo object to a parameter BirthInfo object. The comparison will be alphabetical depending on *only* the Name field. The compareTo method of class BirthInfo should invoke the compareTo method in class Name with the Name instance variable as invoking object and the Name instance variable of the parameter as the parameter for Name's compareTo, and it should return the value that the compareTo method in class Name returns to it.

Specifications for the class BirthInfoList:

An object of class BirthInfoList will have **three** fields. The first field will be an array of BirthInfo objects. The second field will be an int holding the number of records currently in the BirthInfoList. The third field will be a static final int that indicates the maximum capacity of the BirthInfoList.

The BirthInfoList class will have a zero parameter constructor that constructs an array of references to BirthInfo objects of default length equal to the static final int described above. The constructor also initializes the number of records instance variable to zero.

The BirthInfoList class will *at minimum* have the **7** following methods:

a) A boolean valued method named add that takes a BirthInfo parameter representing a person to insert into the database. If a record with the same Name attribute as the parameter is *not* already in the database, the method returns true *after* inserting the new record. If a record with the same Name as the parameter is already in the database, the method returns false and that original record remains unchanged. false is also returned if the list is full.

b) A String valued method named toString that returns a String representation of the entire array of BirthInfo objects.

c) A boolean valued method named isFull that returns true if the list is full and cannot hold another record.

d) A method named alphaSort that sorts the array into alphabetical order.

e) A method named dateSort that sorts the array into the chronological order described above.

f) A BirthInfo valued method named search that has a BirthInfo parameter and returns a reference to the object in the array that matches the Name instance variable of the parameter or returns null if there is no record with matching Name in the list.

g) A boolean valued method named delete that has a BirthInfo parameter and attempts to delete a record from the list that matches the Name instance variable of the parameter. delete returns true if a record is deleted or returns false if no matching record is found to delete, causing the deletion to fail.

Exercise 11.12: This exercise has as its goal the creation of an application program that allows a user to maintain a simple file oriented database of birthday information. A user of your database application should have the following commands available to manipulate the database:

- **Add**: puts information about a person and her/his birthday in the database. At minimum, the user must provide a first name, middle family name, month of birth, and day in that month of birth. The program should NOT proceed without minimal information. The year of birth is an optional piece of data when information is being added to the database. As the programmer, you will need to determine how to code the situation where the year of birth and/or the middle name are unknown so that this situation does not cause your code to fail in the context of other records where the year of birth is known. All of the information taken together for a person shall be referred to as a "record". If a person with the same name already exists in the database, a message should be displayed informing the user about the existing record and the addition of the new record should fail. The program should also display a message to the user after each successful addition of a new record.
- **Display**: outputs the entire contents of the database.
- **Alphabetical Sort and Display**: sorts the database in alphabetical order and then displays all records. Records should be sorted with the person's family name as the primary key, the first name as the secondary key, and the middle name as third key.
- **Chronological Sort and Display**: sorts the database in date order. Dates should be sorted with the month of birth as the primary key, the day in that month of birth as the secondary key, and the optional year of birth the last key. In the case of two entries with the same birth date, sorting will continue with the family name, the first name, and finally the middle name of the person.
- **Search**: allows the user to enter a person's name (family, first, and middle) to determine if the person is in the database. A search cannot proceed without the minimal information of a family name, a first name, and a middle name. If the person is found, her/his information should be displayed. If the person is not found, an appropriate message should be given to the user.
- **Delete**: allows the user to remove the information concerning a specified person from the database. The user should specify the first name, the middle name, and the family name of the person to be deleted. If the person is not found in the database, an appropriate message should be given to the user. Otherwise, once the record is found, it should be displayed, and the user should be asked to confirm the deletion request. The database application should give the user an opportunity to cancel the deletion request. If the user confirms the deletion request, the record should be deleted, and the user should be informed the deletion took place. If the user cancels the deletion request, the user should be informed that the record was not deleted.

1) Create a GUI for the database application in a class named `Database`. Your GUI should include buttons for each of the possible commands: Add, Display, Alphabetical Sort And Display, Chronological Sort and Display, Search, and Delete (including confirming or canceling a delete request). It should also contain text fields and/or combo boxes for user input and a way to display output from the database. At this point the response to any button click should be to display a message that the particular button was clicked or a combo box element was selected.

(continues)

Exercise 11.12, continued

2) Write enough code in your `Database` class so that the buttons for Add and Display in the GUI are functional. In particular, make sure the user provides at least the minimum input required, and that values for dates are valid (30 days hath September, etc).

3) Complete the database application. This means that the Search, Alphabetical Sort And Display, Chronological Sort And Display, and Delete functions of the GUI should work. The methods `alphaSort`, `dateSort`, `search`, and `delete` of the `BirthInfoList` class support those functions of the GUI. Again, make sure input from the user is appropriately error-checked for the new functionality being implemented. Remember to test your program thoroughly. Include a test where the user asks for a deletion and cancels the request after the record is found. Also include a test where the user fills the list, then deletes data, and then fills the list again.

Lab: Arrays and `While` Loops • Chapter 11

The objectives of this lab is:

- to get experience working with arrays and `while` loops

This lab uses a Java class named `ArraysAndWhileLoops` in a code package entitled `for_arrays_and_while_loops_lab`. We provide an application program without a GUI. The code immediately below generates 100 random int values in the range 0–49 and stores them in an array.

You will

- write static methods for the class `ArraysAndWhileLoops`
- write code to test your methods by calling them from within the `main` method.
- write a comment before each method giving the numbered step as indicated below
- write comments in the `main` method indicating the method invocations needed to accomplish each numbered step

To make your output easier to follow

- use `System.out.println` in the `main` method to display sentences describing the output that follows.
- present methods in the order they appear below in the `.java` file.

You may use additional variables and syntactic constructs as needed in conjunction with `System.out.println` to show that your methods work correctly. If a task requires a search, make sure that your tests include tests that you know will fail to find the specified item. Your code in the `main` method should include output that clearly indicates when a search fails. Note that if a task described below indicates that a value should be returned, the method's return type should **not** be void, and your test should use the returned value in some way. However, if a task does not specify that a value is returned, the method's return type should be void.

1) Get a copy of the folder `for_arrays_and_while_loops_lab` onto the **Desktop**.
2) Launch Eclipse.
3) Create a new Java project named with `ArraysAndWhileLoops` **and** `your name`.
4) Drag the folder `for_arrays_and_while_loops_lab` from the **Desktop** onto the image of the your `ArraysAndWhileLoops` folder in the `Package Explorer` view.
5) Double click on your `ArraysAndWhileLoops` folder in the `Package Explorer` view to open the folder. You should see that the package included in your folder is named `for_arrays_and_while_loops_lab`.
6) Double click on the `for_arrays_and_while_loops_lab` package in the `Package Explorer` view to open the folder. You should see that the file named `ArraysAndWhileLoops.java` is included in your package named `for_arrays_and_while_loops_lab`.

7) Double click on ArraysAndWhileLoops.java to open the file in the editor view.

8) Run the program.

9) To make your name appear on the output, use System.out.println in the main method to display your name before any other the output.

10) Write and test a static method that displays the elements in the array myInts until their sum exceeds an int parameter. You should display the element that causes the sum to exceed the parameter.

11) Write and test a static method that determines and returns the index that is the first occurrence of an int parameter in the array myInts. Your method should return –1 if no element in the array is found to be equal to the parameter.

12) Write and test a static method that determines and returns the index of the first occurrence of a value in the array myInts that is greater than an int parameter. Your method should return –1 if no element in the array is found to be greater than the parameter.

13) Write and test a static method that determines and returns the index that is the last occurrence of an int parameter in the array myInts. Your method should return –1 if no element in the array is found to be equal to the parameter.

14) Write and test a static method that determines and returns the index of the last occurrence of a value in the array myInts that is greater than an int parameter. Your method should return –1 if no element in the array is found to be greater than the parameter.

15) Assuming that the array myInts has **not** been sorted, determines and returns the index of the first element of the array that is not in ascending order. Your method should return –1 if the all of the elements of the array are found to be in ascending order.

16) Run the complete program.

17) Copy the output in the Console view and paste it into a word processing document.

18) Print a hard copy of your output.

19) Print the file ArraysAndWhileLoops.java.

20) Submit your hard copies of the code and the output to the lab instructor.

21) Drag a copy of your ArraysAndWhileLoops folder from the **Desktop** onto a flash drive or a file server.

22) In the Package Explorer view select your ArraysAndWhileLoops folder.

23) Select **Edit > Delete**.

24) In the dialog box that opens, click the radio button that indicates you want to delete everything before you click the **Yes** button.

25) Quit Eclipse.

12

Introduction to Abstract Data Types and Java Generics

INTRODUCTION

In Chapter 11, we described a class that uses an array. The class `NameList` maintains a list of `Name` objects. We also discussed several algorithms for the actions of searching and sorting the list. Suppose we were to write a class named `StudentRecord` that represents all of the information about an individual student at our school. We could then write a `StudentRecordList` class that maintains a list of `StudentRecord` objects. How would the two list classes differ? We will examine the differences below and show that the `StudentRecordList` class would be very similar to the `NameList` class. In fact, no matter what class of object is used for the individual elements of a list, the class that represents a list of objects is very similar to our `NameList` class. The only real differences have to do with how we determine equality of elements in the list and ordering of list elements.

Java provides us with more than one mechanism for writing a general `List` class that can be customized to hold a specified kind of object. The first mechanism we will discuss is the use of `abstract` classes. We will write an `abstract List` class and extend that `abstract` class to a more specific subclass tailored to the class of the objects in the list. By writing an `abstract` class we do the bulk of the work only once, hence saving a lot of time when writing a new program. An `abstract` class is said to be *reusable*. The subclass of the `abstract` list class will contain only the details that the `abstract` class needs to work with a particular class of object for its elements.

A second mechanism involves creating a list of elements that are assumed to have information about how equality and ordering are determined.

We will show that both of these mechanisms can work, but have a serious problem in that the compiler cannot detect when different kinds of objects that cannot be compared to one another are stored in a list. In this situation, the program can execute without any problems until a comparison is needed. Then a runtime `ClassCastException`

is thrown, making the problem obvious as one object cannot be cast to the class of the other.[1]

The third mechanism we will discuss involves the use of *type parameters* to specify the type of objects in the list. This method gives more information to the compiler so that any attempt in the code to add items to the list that cannot be compared is caught by the compiler. It is more efficient in the world of software development to have problems caught in the compile stage instead of later during execution.

We will also discuss a mechanism for writing a class to represent a list that can use the natural ordering of the objects in the list to sort the list or use an alternative ordering scheme.

Code in the NameList Class That Is Specific to Name Objects

Consider the `Name` class and the `NameList` class displayed below. The bulk of the `NameList` class is general in nature and could be used for a list whose elements are from any class. We have highlighted the code in the `NameList` class that works only for array elements of class `Name`.

```
public class Name implements Comparable<Name> {
  private String myFirst; // reference to the first name
  private String myMiddle; // reference to the middle name
  private String myFamily; // reference to the family name

  public Name (String theFirst, String theMiddle, String theFamily) {
       myFirst = theFirst;
       myMiddle = theMiddle;
       myFamily = theFamily;
  }//3 parameter constructor

  public Name (String theFirst, String theFamily) {
       myFirst = theFirst;
       myMiddle = "";
       myFamily = theFamily;
  }//2 parameter constructor

  public String getFirst() {
       return myFirst;
  }//getFirst

  public void setFirst(String theFirst) {
       myFirst = theFirst;
  }//setFirst

  public String getMiddle() {
    return myMiddle;
  }//getMiddle
```

[1]These first two options and the use of an interface instead of an abstract class were the only options available in versions of Java before 1.5.

```
  public void setMiddle(String theMiddle) {
       myMiddle = theMiddle;
  }//setMiddle

  public String getFamily() {
       return myFamily;
  }//getFamily

  public void setFamily(String theFamily) {
       myFamily = theFamily;
  }//setFamily

  public boolean equals(Object theOther){
       Name theName = (Name)theOther;
       return myFirst.equals(theName.myFirst)
            && myMiddle.equals(theName.myMiddle)
            && myFamily.equals(theName.myFamily);
  }//equals

  public int compareTo(Name theName){
       int compareFamily = myFamily.compareTo(theName.myFamily);
       if(compareFamily != 0) {
            return compareFamily;
       }//family names were different

       int compareFirst = myFirst.compareTo(theName.myFirst);
       if(compareFirst != 0) {
         return compareFirst;
       }//family names were equal and first names were different

       return myMiddle.compareTo(theName.myMiddle);
  }//compareTo

  public boolean before(Name theOther) {
       return compareTo(theOther) < 0;
  }//before

  public String toString() {
       return myFamily + ", " + myFirst +
            (myMiddle.equals("") ? "" : " " + myMiddle);
  }//toString
}//Name

public class NameList {
  private Name myArray[]; //refers to the array of Name objects
  private int myCount; // holds the current count of Name objects
                       // in the list
  public NameList () {
       myArray = new Name [100];
       myCount = 0;
  }// 0 parameter constructor
```

```
    public NameList (int theMax) {
         myArray = new Name [theMax];
         myCount = 0;
    }// 1 parameter constructor

    public boolean isFull() {
         return myCount == myArray.length;
    }//isFull

    public boolean add(Name theName ) {
         if (!isFull()) {
              myArray [myCount++] = theName ;
              return true;
         } // add was successful
         System.out.println("Attempting to add to full array");
         return false;
    }//add

    public Name search (Name theName ) {
         int i = 0;
         while ( i < myCount && !( myArray [i].equals(theName ))){
              i++;
         }//while
         return (i < myCount ? myArray[i] : null);
    }//search

    private void swap (int firstPos, int otherPos) {
         Name temp = myArray [firstPos];
         myArray [firstPos] = myArray [otherPos];
         myArray [otherPos] = temp;
    }//swap

    public void bubbleSort() {
         for(int passNum = 1; passNum < myCount; passNum++) {
              for (int j = 0; j < myCount - passNum; j++) {
                if(myArray [j+1].before (myArray [j])) {
                     swap(j, j+1);
                }//if
              }//inner for loop for one pass
         }//for loop that controls passes over the data
    }//bubbleSort

    public String toString() {
         String stringToReturn = "";
         for (int i = 0; i < myCount; i++) {
           stringToReturn += myArray [i] + "\n"; //"\n" for the new line character
         }//for
         return stringToReturn;
    }//toString

}//NameList
```

Changing the NameList Class into a General List Class by Replacing Name with Object

We can make a list of the critical places in the code of the NameList class where the Name class is used or where we have used the word "Name" as part of an identifier. We will replace each reference to the Name class with a reference to Java's most general class, the Object class. Additionally, we will modify any identifiers we have created to make them more general.

1) `public class NameList {`

   The identifier used to name the class is NameList. To make a general class, we could choose the identifier List instead.

   ```
   public class List {
   ```

2) `private Name myArray[];`

   myArray is declared to be an array of references to Name objects. To make the array hold general elements, we can declare myArray to be an array of references to Objects, where Object refers to the class at the top of Java's class hierarchy.

   ```
   private Object myArray[];
   ```

3) `myArray = new Name [100];` or `myArray = new Name [theMax];`

   myArray is constructed as an array of Name references. Instead we construct an array of Object references.

   ```
   myArray = new Object [100]; or myArray = new Object [theMax];
   ```

4) In the add method we have a parameter named theName of class Name.

   `public boolean add(Name theName) {`

   We can change the parameter to addMe is of class Object.

   ```
   public boolean add(Object addMe) {
   ```

5) In the search method, again there is a parameter named theName of class Name. Additionally, the method returns a reference of class Name. We change the parameter to findMe of class Object. We also change the return type of the method to class Object.

   ```
   public Name search (Name theName) {
   ```

   becomes

   ```
   public Object search (Object findMe) {
   ```

   In addition there is an instance of code that invokes the equals method of the Name class

   ```
   myArray[i].equals(findMe)
   ```

Here Java's inheritance and dynamic binding of methods does exactly the correct thing!

> From the perspective of the compiler, there is reference to an `Object` invoking its `equals` method with an argument that is a reference to an `Object`. This syntax matches the signature of the `equals` method in the `Object` class. Hence, the compiler finds this code to be correct. At run time, Java's *dynamic binding* looks **not** at the *declared type* of `myArray[i]`, but at the *actual type* of the object assigned to `myArray[i]` in order to determine the appropriate `equals` method to execute. Since we intend to add `Name` objects to our list, the actual type of `myArray[i]` will be `Name` and the `equals` method of the `Name` class will be invoked. This will be true in general for lists of any particular kind of object as long as the class correctly overrides the `equals` method of the `Object` class.

6) In the `swap` method, the `temp` variable used to implement the `swap` is declared to be a reference to an object of class `Name`

```
Name temp = myArray[firstPos];
```

We will simply declare the `temp` variable to be a reference to an object of class `Object` as follows

```
Object temp = myArray[firstPos];
```

7) In the `bubbleSort` method, there is code that invokes the `before` method in the `Name` class. There is no `before` method in the `Object` class. Hence, code like `myArray[j+1].before(myArray[j])` will not compile when `myArray[j+1]` is declared to be a reference to an `Object`.

Because the `Object` class does not have a `before` method, we must re-write this method invocation so that the method is **not** invoked on an `Object`. Instead we will write a method of the `List` class that is passed references to the two `Object`s being compared. Note that the syntax changes slightly, but in an important way because the `before` method now becomes a method of the class in which it is invoked, and we see it invoked with two arguments instead of one invoking object reference and one argument.

```
if(myArray[j+1].before(myArray[j]) {
```

becomes

```
if(before(myArray[j+1], myArray[j])) {
```

when the `bubbleSort` method needs to determine if one element of the list is before another element. However, we do not want to provide the details of the `before` method. *Those details are dependent on the type of* `Object` *that is added to the* `List`. Hence, we provide a return type and method signature, but no method body, and we indicate to the Java compiler that the body will be provided when

the programmer **extends** this class by indicating that the method is **abstract**. The **abstract before** method is declared as follows:

```
public abstract boolean before (Object obj1, Object obj2);
```

Note that this looks like the beginning of a method, except for the crucial difference that the line ends with a semicolon instead of an opening curly bracket.

This **abstract** method forces our general **List** class to also be **abstract**. We will not be able to construct objects of our **abstract List** class. Within a subclass that extends **List** we will supply the details of the definition of the **before** method.

8) Consider the **toString** method in the **NameList** class.

```
public String toString() {
  String stringToReturn = "";
    for (int i = 0; i < myCount; i++) {
      stringToReturn += myArray[i] + "\n"; //"\n" for the new line character
    }//for
  return stringToReturn;
}//toString
```

What happens when the line of code

```
stringToReturn += myArray[i] + "\n";
```

is executed and **myArray[i]** has declared type **Object** and actual type **Name**? The answer is that the **+** implicitly causes a **toString** method to be executed. Since there is a **toString** method in the **Object** class the compiler finds no problem. The **toString** method in the **Name** class is executed due to Java's dynamic binding. The declared class, **Object**, has a **toString** method that satisfies the compiler and the actual class, **Name**, has a **toString** method that is invoked when the code is executed.

Mechanism 1 for Writing a General List Class: the abstract List Class

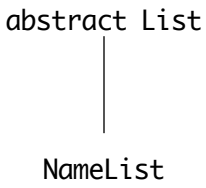
Below we have assembled the **abstract List** class.

```
public abstract class List  {

  private Object  myArray[]; //refers to the list of Object  objects
  private int myCount; // holds the current count of objects
                       // in the list

  public List () {
       myArray = new Object  [100];
       myCount = 0;
  }// 0 parameter constructor

  public List (int theMax) {
    myArray = new Object  [theMax];
    myCount = 0;
```

```
   }// 1 parameter constructor

   public boolean isFull() {
      return myCount == myArray.length;
   }//isFull

   public boolean add(Object  addMe ){
      if (!isFull()) {
        myArray[myCount++] = addMe ;
        return true;
      } // add was successful
      System.out.println("Attempting to add to full array");
      return false;
   }//add

   public Object  search (Object  findMe  ) {
      int i = 0;
      while ( i < myCount && !(myArray[i].equals(findMe  ))){
            i++;
      }//while
      return (i < myCount ? myArray[i] : null);
   }//search

   private void swap (int firstPos, int otherPos) {
      Object  temp = myArray[firstPos];
      myArray[firstPos] = myArray[otherPos];
      myArray[otherPos] = temp;
   }//swap

   public void bubbleSort() {
      for(int passNum = 1; passNum < myCount; passNum++) {
        for (int j = 0; j < myCount - passNum; j++) {
          if(before  (myArray[j+1], myArray[j])  ) {
            swap(j, j+1);
          }//if
        }//inner for loop for one pass
      }//for loop that controls passes over the data
   }//bubbleSort

   public String toString() {
      String stringToReturn = "";
      for (int i = 0; i < myCount; i++) {
        stringToReturn += myArray[i] + "\n";//"\n" for the new line character
      }//for
      return stringToReturn;
   }//toString

   public abstract boolean before (Object obj1, Object obj2);
}//List class
```

Using the abstract List Class by Defining a Concrete Subclass NameList

We can never construct a **List** object from this class definition because the definition is incomplete in that it has an **abstract** method. We can, however, extend the class to a concrete subclass named **NameList** from which objects can be constructed if the subclass provides a definition for the **abstract before** method. In terms of inheritance, any **NameList** object is also a **List** object. We read the following diagram from bottom to top inserting the words "is a" or "is an" as you travel up a line.



We will define a new version of class **NameList** below. In essence, the new version of the **NameList** class provides a bridge to the **before** method in the **Name** class. It is in the **NameList** class that our elements of class **Object** must be recognized as elements of class **Name**. We will use the Java facility of type casting to a class to accomplish the identification of the appropriate class of an object.

```
public class NameList extends List {
    public NameList() {
        super(); //invokes the 0 parameter constructor in the List class
    }// 0 parameter constructor

    public NameList(int theMax) {
        super(theMax); //invokes the 1 parameter constructor in the List class
    }// 1 parameter constructor

    public boolean before (Object obj1, Object obj2) {
        return ((Name) obj1).before((Name) obj2);
    }// before
}// class NameList
```

The Constructors of the NameList Class

Notice that our new version of the **NameList** class has two constructors: a 0 parameter constructor and a 1 parameter constructor. These correspond to the 0 parameter and 1 parameter constructors of the **abstract List** class.

The only line of code in the 0 parameter constructor is

```
super();
```

This line of code invokes the 0 parameter constructor in the **super** class of **NameList**, the **abstract List** class.

The only line of code in the 1 parameter constructor is

```
super(theMax);
```

This line of code invokes the 1 parameter constructor in the **super** class of **NameList**, the **abstract List** class, passing it the **int** parameter **theMax**.

When a NameList object is constructed, it *inherits* all of the instance variables and methods of the List class since NameList extends List. In addition, the NameList object has a concrete before method.

We will trace the flow of execution of the following lines of code when you declare a NameList reference and then construct an object of class NameList to which that reference points:

```
NameList myListOfNames;
myListOfNames = new NameList(125);
```

The 1 parameter NameList constructor has been called. It executes its only line of code which invokes the 1 parameter constructor of the abstract List class.[2] This super class constructor constructs the array of Object references of the specified size and assigns it to the instance variable myArray. It also intializes the instance variable myCount to 0.

Exercise 12.1: Describe the flow of execution for the following lines of code.

```
NameList myListOfNames;
myListOfNames = new NameList();
```

Note this rule in Java: If a constructor invokes its super class constructor, the invocation must be its first instruction. We have seen that situation many times in our applications that extend the JFrame class. We have written

```
super("Title to appear in title bar ");
```

Casting an Object to Another Class in the Comparison Methods of Class NameList

Consider the before method in our NameList class. The signature of the method must exactly match the signature of the abstract before method in the abstract List class. Hence, the two parameters obj1 and obj2 are of class Object making the header of the method the following:

```
public boolean before (Object obj1, Object obj2) {
```

We need to provide the means for comparison. We know that our Name objects have a before method that can do the comparison. Java can only invoke that before method if it has two Name objects, one to invoke the method, and one to pass as an argument to the method. Hence, when we invoke the before method, we need to inform Java of the actual types of the objects. We will use obj1 as the invoking object. To indicate to Java the actual type of obj1, we perform a type cast syntactically by placing the name of the class to which we are casting in parentheses in front of the reference to the object being cast (in this case obj1).

```
(Name) obj1
```

In order to use this type cast in our invocation of the before method, we enclose that expression in parentheses before using the dot and method name to invoke the method because the rules of precedence in Java give higher precedence to dot than to casting.

[2]The constructor of the abstract List class invokes the constructor of its super class, Object, implicitly. Since this happens for during the construction of all Java objects, we do not dwell on it here.

```
((Name) obj1).before(. . .
```

In other words, the following code will not compile

```
(Name) obj1.before(. . .//this won't work
```

because Java sees it as though it was parenthesized as follows:

```
(Name) (obj1.before(. . .)) //this won't work
```

Since there is no **before** method in the **Object** class, this will not work. And even if there were such a method, the resulting **boolean** value could not be cast to the class **Name**.

The **before** method in class **Name** takes one argument of class **Name**. Hence, **obj2** must also be cast to class **Name** before it is used as an argument, resulting in the following code:

```
((Name) obj1).before((Name) obj2)
```

The complete method **before** of class **NameList** is as follows:

```
public boolean before (Object obj1, Object obj2) {
    return ((Name) obj1).before ((Name) obj2);
}// before
```

Using the Methods of the NameList Class

Suppose we have declared and constructed **myListOfNames** as described above using the 0 parameter constructor. Now we would like to put a **Name** in the list using the **add** method. The **add** method of the **NameList** class which is inherited from the **abstract List** class has the following header:

```
public void add(Object  addMe) {
```

The parameter **addMe** is a reference to an **Object**. As far as the compiler is concerned, we can add any type of object to the list. However, our program will only run successfully with **Name** objects. In order to add Barbara Ann-Teresa Li Santi to the list, we first need to construct a **Name** object and then pass it as an argument to the **add** method of **myListOfNames**. Recall that the **add** method has **boolean** return type. Hence, our code below prints an appropriate message depending on the value returned by the **add** method.

```
Name oneName = new Name("Barbara", "Ann-Teresa", "Li Santi");
if(myListOfNames.add(oneName)){
    System.out.println(oneName.toString() + " added successfully.");
}else {
    System.out.println(oneName.toString() + " not added.");
}//else
```

The parameter **addMe** of the **add** method is assigned the value of the corresponding argument, **oneName**, which is a reference to an object of class **Name**. This assignment is allowed under the inheritance rules of Java because class **Object** is implicitly the super-class of any class.

The identifier `addMe` has *declared type* `Object` and *actual type* `Name`. The declared type of a reference is the class name used in the declaration. The actual type of a reference is the class from which its object was constructed. This kind of difference is legitimate in Java and in fact, makes many inheritance concepts work. The rule to remember is that it is syntactically and semantically correct if a ***superclass reference points to a subclass object***. That rule describes exactly what happens when the object of class `Name` is passed as an argument to a parameter of class `Object`. In this example, the declared class of every reference in the array is `Object` and the actual class of every object pointed to by each reference is `Name`.

As noted above in the discussion of the `before` method, we sometimes need to resurrect the actual class of an object in order to invoke a method of that class. We use casting to the appropriate class in those situations. In those situations, the cast is down the class hierarchy. In general, casting down the hierarchy is viewed as unsafe casting because the cast could be inappropriate. We have specifically cast down the hierarchy when we know it to be safe.

For an example of an unsafe, inappropriate cast down the hierarchy, consider the next example. The example below shows attempt to cast an object whose actual type is `String` to a `Name` object.

```
String myStr = new String("Hello");
Object myObj = myStr; // references of class Object can point to objects of
                      // any class
Name myName = (Name) myObj; //This is a bad cast
```

The class `String` is also a subclass of class `Object`. The cast of `myObj` to class `Name` is inappropriate because the actual class `myObj` is `String` and a `String` object cannot be cast to a `Name` object.

In general, if the compiler can determine that a cast is impossible, it will generate an error message and the code will not compile. However, in certain circumstances, it is only at run time that a cast is found to be inappropriate and a `ClassCastException` is thrown.

More about Dynamic Binding of Methods to Classes

Consider the following diagram:

In this diagram, **Widget** is a subclass of **Object**, and **Gadget** is a subclass of **Widget**. Each of these classes may have a method with the same signature. When a superclass has a method with the same signature as one of its subclasses, the Java Virtual Machine begins at the actual class of the object and searches upward through the hierarchy for the first occurrence of the method to execute. For example, assume we have an object whose declared class is **Object** and whose actual class is **Gadget**. Also assume that the class **Widget** has a **toString** method, but the class **Gadget** does not. Assume that we invoke the **toString** method on our object. Java does not give us an error message at compile time because the class **Object** has a **toString** method. At execution time, Java will begin by looking for a **toString** method in the actual class of the object, which is the **Gadget** class. When it does not find it there, it will then look in the **Widget** class. Since it finds it there, it will use the **toString** method belonging to the **Widget** class.

In technical terms the binding of a method to a class is *dynamic*, i.e., it happens at run time. If we had not provided a **toString** method in the **Widget** class, the **toString** method in the **Object** class would be executed. Since the **toString** method in the **Object** class returns the class name and something that looks like the address of the object, it probably would not be what we expected.

Testing the NameList class

A small test program for our **NameList**class is displayed below. Note the use of the **boolean** value returned by the **add** method to select an appropriate confirmation message and implicit invocations of the **toString** method.

```
public class TestNameList {
  public static void main(String[] args) {
    NameList myListOfNames = new NameList(100);
    if(myListOfNames.add(new Name("Elizabeth","Catherine","Li Santi"))){
      System.out.println("Elizabeth Catherine Li Santi added successfully.");
    }else {
      System.out.println("Elizabeth Catherine Li Santi not added.");
    }//else
    if(myListOfNames.add(new Name("Lydia","Michelle","Mann"))){
      System.out.println("Lydia Michelle Mann added successfully.");
    }else {
      System.out.println("Lydia Michelle Mann not added.");
    }//else
    if(myListOfNames.add(new Name("Barbara","Ann-Teresa","Li Santi"))){
      System.out.println("Barbara Ann-Teresa Li Santi added successfully.");
    }else {
      System.out.println("Barbara Ann-Teresa Li Santi not added.");
    }//else
    System.out.println("The list is:\n" + myListOfNames);
    myListOfNames.bubbleSort();
    System.out.println("The sorted list is:\n" + myListOfNames);
    if(myListOfNames.search(new Name("Lydia","Michelle","Mann")) != null){
```

```
        System.out.println("Lydia Michelle Mann found.");
      }else {
        System.out.println("Lydia Michelle Mann not found.");
      }//else
      if(myListOfNames.search(new Name("Lydia","Mann")) != null){
        System.out.println("Lydia Mann found.");
      }else {
        System.out.println("Lydia Mann not found.");
      }//else
    }//main
}//TestNameList
```

Below is a transcript of execution of this test program.

```
Elizabeth Catherine Li Santi added successfully.
Lydia Michelle Mann added successfully.
Barbara Ann-Teresa Li Santi added successfully.

The list is:
Li Santi, Elizabeth Catherine
Mann, Lydia Michelle
Li Santi, Barbara Ann-Teresa

The sorted list is:
Li Santi, Barbara Ann-Teresa
Li Santi, Elizabeth Catherine
Mann, Lydia Michelle
Lydia Michelle Mann found.
Lydia Mann not found.
```

Note that the definition of the `equals` method of the `Name` class causes the names "Lydia Michelle Mann" and "Lydia Mann" to be considered unequal.

Mechanism 2—Writing a General List Class of Comparable Objects

The method that forced our `List` class to be abstract was the `before` method. Since there is no `before` method in the `Object` class, the compiler would not accept

```
myArray[j + 1].before(myArray[j])
```

However, if we look carefully at the implementation of the `before` method in the `Name` class, we see that the `before` method is dependent on the `compareTo` method of the `Name` class. The `compareTo` method of the `Name` class allows the class to implement the `Comparable<Name>` interface. We could rewrite the `bubbleSort` method of the class `NameList` as follows using the `compareTo` method instead of the `before` method.

```
public void bubbleSort() {
        for(int passNum = 1; passNum < myCount; passNum++) {
                for (int j = 0; j < myCount – passNum; j++) {
                        if(myArray [j+1].compareTo(myArray [j]) < 0) {
                                swap(j, j+1);
                        }//if
                }//inner for loop for one pass
        }//for loop that controls passes over the data
}//bubbleSort
```

We *cannot* use this code in our `List` class as written above because the compiler will not find a `compareTo` method in the `Object` class. However, we can make a general class of `Comparable` objects where the declared type of each array element assures the compiler that an invocation of `compareTo` is appropriate. To do this we replace every occurrence of `Object` with `Comparable` and arrive at the second version of class `List` below.

```
public class List {
     private Comparable  myArray[]; // array of Objects
     private int myCount;// current count of Objects in List
     public List () {
          myArray = new Comparable [100];
          myCount = 0;
     }//0 parameter constructor
     public List (int theSize) {
          myArray = new Comparable [theSize];
          myCount = 0;
     }//1 parameter constructor

     public boolean isFull(){
          return myCount == myArray.length;
     }//isFull

     public boolean add(Comparable  addMe) {
          if (!isFull()){
               myArray[myCount++] = addMe;
               return true;
          }//add was successful
          System.out.println("Attempting to add to full array");
          return false;
     }//add

     public Comparable  search (Comparable  findMe){
          int i = 0;
       while (i < myCount && !(myArray[i].equals(findMe))){
            i++;
       }//while
       return (i < myCount ? myArray[i] : null);
     }//search
```

```
    private void swap (int firstPos, int otherPos){
        Comparable  temp = myArray[firstPos];
        myArray[firstPos] = myArray[otherPos];
        myArray[otherPos] = temp;
    }//swap

    public void bubbleSort(){
        for (int passNum = 1; passNum < myCount; passNum++) {
          for(int j = 0; j < myCount - passNum; j++) {
              if(myArray[j + 1].compareTo( myArray[j]) < 0    ){
                swap(j, j + 1);
              }//if
          }//inner for
        }//outer for
    }//bubbleSort

    public String toString(){
        String stringToReturn = "";
        for (int i = 0; i < myCount; i++){
          stringToReturn += myArray[i] + "\n";//"\n" for the new line character
        }//for
        return stringToReturn;
    }//toString
}//List
```

The compiler issues no errors for this code since `myArray[j + 1]` is declared to be a reference to a `Comparable` object and hence can invoke a `compareTo` method, but does issue a warning: `Type Safety: The method compareTo(Object) belongs to the raw type Comparable. References to generic type Comparable<T> should be parameterized.` This error message is telling the programmer to use the type parameters that became available with Java 1.5. We will do this when we describe the third mechanism for writing a general `List` class.

Recall the header of the `compareTo` method that we wrote in the `Name` class:

```
public int compareTo(Name theName){
```

The declared type of `myArray[j + 1]` is `Comparable`, but in our example the actual type will be `Name`. Since the actual type determines where the runtime system starts its search for the method to execute, the `compareTo` method of the `Name` class will be executed. The `compareTo` method of the `Name` class has a `Name` parameter. In general, we cannot assign an `Object` to a `Name` reference because an `Object` is not necessarily a `Name`. The compiler cannot know the actual class of the invoking object. Hence, it issues a warning that the types may not match at the moment that the argument is assigned to the parameter.

We can use our class `TestNameList` above to test this second version of class `List`.

The Problems Involved with the First Two Mechanisms

Both of the mechanisms discussed so far for abstracting the concept of a list suffer from a dangerous pitfall. Different classes of `Object`s can be added to our first version of a `List` object and different classes of `Comparable`s can be added to our second version of a `List` object without any complaint from the compiler or runtime system. However, if our test program attempts to sort the list, using the first mechanism, the program will throw a `ClassCastException` generated by the `before` method of class `NameList`. Using the second mechanism, our program will throw a `ClassCastException` generated by the `compareTo` method in the class of the invoking object. In addition, if a test program attempts to search for an object of a class different from the objects in the list, the program will throw a `ClassCastException` generated by the `equals` method in the class of the invoking object. In any of these cases, the program cannot continue to execute in a consistent manner.

The program below shows the problem generated by adding three `Name`s and one `String` to a `NameList` that extends `abstract List` object and then attempting to sort the `NameList`.

```
public class TestNameList {
  public static void main(String[] args) {
    NameList myListOfNames = new NameList(100);
    if(myListOfNames.add(new Name("Elizabeth", "Catherine", "Li Santi"))){
      System.out.println("Elizabeth Catherine Li Santi added successfully.");
    }else {
      System.out.println("Elizabeth Catherine Li Santi not added.");
    }//else
    if(myListOfNames.add(new Name("Lydia", "Michelle", "Mann"))){
      System.out.println("Lydia Michelle Mann added successfully.");
    }else {
      System.out.println("Lydia Michelle Mann not added.");
    }//else
    if(myListOfNames.add(new Name("Barbara", "Ann-Teresa", "Li Santi"))){
      System.out.println("Barbara Ann-Teresa Li Santi added successfully.");
    }else {
      System.out.println("Barbara Ann-Teresa Li Santi not added.");
    }//else
    //*********************************************************************
    // Here a String is added to a list of Name objects
    if(myListOfNames.add("Hello")){
      System.out.println("String Hello added successfully.");
    }else {
      System.out.println("String Hello  not added.");
    }//else
    //*********************************************************************
    System.out.println("The list is:\n" + myListOfNames.toString());
    myListOfNames.bubbleSort();
    System.out.println("The sorted list is:\n" + myListOfNames.toString());
  }//main
}//TestNameList
```

Below is a transcript of the execution that is terminated by the ClassCastException.

```
Elizabeth Catherine Li Santi added successfully.
Lydia Michelle Mann added successfully.
Barbara Ann–Teresa Li Santi added successfully.
String Hello added successfully.

The list is:
Li Santi, Elizabeth Catherine
Mann, Lydia Michelle
Li Santi, Barbara Ann–Teresa
Mann, George Millard
Hello
```

The error message below describes the method invocations on the runtime stack at the time that the ClassCastException was thrown. We read it from the bottom upward.

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
    at NameList.before(NameList.java:14)
    at List.bubbleSort(List.java:48)
    at TestNameList.main(TestNameList.java:33)
```

Line 33 of the main method of TestNameList.java is:

```
myListOfNames.bubbleSort();
```

Line 48 of the bubbleSort method of List.java is:

```
if(before(myArray[j + 1], myArray[j])){
```

Line 14 of the before method of NameList.java is:

```
return ((Name) obj1).compareTo((Name)obj2) < 0;
```

obj1 is a String and cannot be cast to a Name. Hence, a ClassCastException is thrown.

Using the second version of class List in the code above, we obtain the following transcript of execution:

```
Elizabeth Catherine Li Santi added successfully.
Lydia Michelle Mann added successfully.
Barbara Ann–Teresa Li Santi added successfully.
String Hello added successfully.

The list is:
Li Santi, Elizabeth Catherine
Mann, Lydia Michelle
Li Santi, Barbara Ann–Teresa
Hello

Exception in thread "main" java.lang.ClassCastException: Name
    at java.lang.String.compareTo(String.java:90)
    at List.bubbleSort(List.java:47)
    at TestNameList.main(TestNameList.java:33)
```

Line 33 of the main method of TestNameList.java is:

```
myListOfNames.bubbleSort();
```

Line 47 of the `bubbleSort` method of our second version of `List.java` is:

```
if(myArray[j + 1].compareTo(myArray[j]) < 0){
```

We do not have the source code for Java's String class, but we can still deduce the problem. Line 90 of the `compareTo` method of `String.java` expects the parameter of a `compareTo` to be able to be cast to a `String`. However, a `Name` object cannot be cast to a `String`.

In the following test program, we attempt to search for the `String "Hello"` in a list of `Name`s using the `abstract List` class mechanism.

```
public class TestNameList {
  public static void main(String[] args) {
    NameList myListOfNames = new NameList(100);
    if(myListOfNames.add(new Name("Elizabeth", "Catherine", "Li Santi"))){
      System.out.println("Elizabeth Catherine Li Santi added successfully.");
    }else {
      System.out.println("Elizabeth Catherine Li Santi not added.");
    }//else
    if(myListOfNames.add(new Name("Lydia", "Michelle", "Mann"))){
      System.out.println("Lydia Michelle Mann added successfully.");
    }else {
      System.out.println("Lydia Michelle Mann not added.");
    }//else
    if(myListOfNames.add(new Name("Barbara", "Ann-Teresa", "Li Santi"))){
      System.out.println("Barbara Ann-Teresa Li Santi added successfully.");
    }else {
      System.out.println("Barbara Ann-Teresa Li Santi not added.");
    }//else
    System.out.println("The list is:\n" + myListOfNames.toString());
//*************************************************************************
    if(myListOfNames.search("Hello") != null){
      System.out.println("Hello found.");
    }else {
      System.out.println("Hello not found.");
    }//else
  }//main
}//TestNameList
```

Below is a transcript of the execution that is terminated by a `ClassCastException`.

```
Elizabeth Catherine Li Santi added successfully.
Lydia Michelle Mann added successfully.
Barbara Ann-Teresa Li Santi added successfully.
String Hello added successfully.
The list is:
```

```
Li Santi, Elizabeth Catherine
Mann, Lydia Michelle
Li Santi, Barbara Ann-Teresa
Hello

Exception in thread "main" java.lang.ClassCastException: java.lang.String
    at Name.equals(Name.java:43)
    at List.search(List.java:33)
    at TestNameList.main(TestNameList.java:36)
```

Line 36 of the `main` method of `TestNameList.java` is:

```
if(myListOfNames.search("Hello") != null){
```

Line 33 of the `search` method of `List.java` is:

```
while (i < myCount && !(myArray[i].equals(findMe))){
```

Line 43 of the `equals` method of `Name.java` is:

```
Name theName = (Name)theOther;
```

`theOther` is a `String` and cannot be cast to a `Name`. Hence, a `ClassCastException` is thrown.

Using the second version of class `List` in the code above, we obtain the following transcript of execution:

```
Elizabeth Catherine Li Santi added successfully.
Lydia Michelle Mann added successfully.
Barbara Ann-Teresa Li Santi added successfully.
String Hello added successfully.

The list is:
Li Santi, Elizabeth Catherine
Mann, Lydia Michelle
Li Santi, Barbara Ann-Teresa
Hello

Exception in thread "main" java.lang.ClassCastException: java.lang.String
    at Name.equals(Name.java:43)
    at List.search(List.java:32)
    at TestNameList.main(TestNameList.java:35)
```

Line 35 of the `main` method of `TestNameList.java` is:

```
if(myListOfNames.search("Hello")!= null){
```

Line 33 of the `search` method of `List.java` is:

```
while (i < myCount && !(myArray[i].equals(findMe))){
```

Line 43 of the `equals` method of `Name.java` is:

```
Name theName = (Name)theOther;
```

theOther is a String and cannot be cast to a Name. Hence, a ClassCastException is thrown.

In order to catch this kind of error at compile time, Java 1.5 introduced the concept of type parameters.

Mechanism 3—Writing a General List Class with a Type Parameter, Our First Look at Java's Generics

Our goal is to find a way to write a general list class so that the compiler catches the kinds of errors described above. In Java 1.5, type parameters were added to the language to allow for this kind of compiler error checking. We would like to indicate the class of the elements in a list so that items that are in some way incompatible are caught early by the compiler, instead of late by the runtime system. In our example, we would like to declare that our list will contain elements that are Name objects. The syntax in Java 1.5 allows for a type parameter to be specified for our third version of class List as follows:

```
List<Name> myListOfNames = new List<Name>();
```

The angle brackets < and > are the signal to the complier that Name is acting as a type parameter. Note here that both the declared type and the actual type of myListOfNames is List<Name>. With this declaration the following code will not compile:

```
if(myListOfNames.add("Hello")){//this generates an compile time error
```

With a properly written class List<E>, the line of the above code will generate the following error message, preventing the code from compiling:

```
The method add(Name) in the type List<Name> is not applicable for the
arguments (String).
```

In other words, the add method must be passed a reference that is a Name.

The following line of code also fails to compile in a similar manner:

```
if(myListOfNames.search("Hello") != null){ //this generates a compile
                                                    //time error
```

This line of code will generate the following error message, preventing the code from compiling:

```
The method search(Name) in the type List<Name> is not applicable for the
arguments (String)
```

In other words, the search method must also be passed a reference that is a Name.

It is now our job to write the class List<E extends Comparable<E>>. The convention in Java is to use a single upper case letter to indicate a type parameter. It is also convention to use E (for element) for the type parameter since they are so frequently used in the API supplied data structures to stand for the kind of *element* being stored in the data structure.

In our example, we want the elements in the list to be Comparable. Java allows us to add such restrictions to the type parameter. Instead of using the word implements

as we have with classes that implement the `Comparable<E>` interface, we use the word `extends` for type parameters. In addition, as we have seen with the `Name` class in Chapter 11, the `Comparable` interface itself has a type parameter. Our `Name` class implements `Comparable<Name>` just as the `String` class implements `Comparable<String>`. In order to have our `List` class restrict elements in the list to a class of object that `implements` the `Comparable<E>` interface, we write the class header as follows:

```
public class List <E extends Comparable<E>> {
```

Since our class `Name` implements the `Comparable<Name>` interface, the compiler will allow the syntactic construct `List<Name>`. It is as though `Name` takes the place of `E` in the class definition of `List<E extends Comparable<E>>.`

Since we are using `E` for our generic element type, the declarations of the instance variables look straightforward:

```
private E myArray[]; // array of E references
private int myCount;// current count of Es in List
```

However, the constructors are not so straightforward. It is illegal in Java to construct an object of a type parameter or to construct an array of references to type parameter objects. Hence, the following code will not compile:

```
E oneE = new E()// illegal—you cannot construct an E
E[] arrayOfE = new E[100]; // illegal—you cannot construct an array of E
```

A full discussion of the reasons for this prohibition is beyond the scope of this text. Please, see the Java Programming Language, 5th Edition, if you are curious and ready for the details!

In writing our `List<E extends Comparable<E>>` class, we get around this prohibition with code that generates a warning, but works. We construct an array of `Comparable` references and then cast that array to `E[]` as follows in our two constructors:

```
public List () {
    myArray = (E []) new Comparable[100];
    myCount = 0;
}//0 parameter constructor

public List (int theSize) {
    myArray = (E []) new Comparable[theSize];
    myCount = 0;
}//1 parameter constructor
```

The warning generated by the type casts is as follows:

```
Type safety: The cast from Comparable[] to E[] is actually checking against the
erased type Comparable[]
```

The warning is saying that the type `E` is really replaced by `Comparable` for the runtime system and hence the cast does not really have much effect.[3] Remember that the whole

[3]This is known as type erasure. Once again, the details are beyond the scope of this text.

point of type parameters is to catch errors at compile time instead of runtime. What happens is that we get to write code without those type casts that cause the runtime errors discussed above. It is true that the casts are still necessary. The compiler puts them in as needed.

The remainder of the class `List<E extends Comparable<E>>` looks just like our second version of class `List` except that everywhere `List` had `Comparable`, class `List<E extends Comparable<E>>` has the type parameter `E`. We have assembled the entire class below:

```
public class List <E extends Comparable<E>>  {

    private E myArray[]; // array of Es
    private int myCount;// current count of Es in List

    public List () {
        myArray = (E []) new Comparable[100];
        myCount = 0;
    }//0 parameter constructor

    public List (int theSize) {
        myArray = (E []) new Comparable[theSize];
        myCount = 0;
    }//1 parameter constructor

    public boolean isFull(){
        return myCount == myArray.length;
    }//isFull

    public boolean add(E addMe) {
        if (!isFull()){
            myArray[myCount++] = addMe;
            return true;
        }//add was successful
        System.out.println("Attempting to add to full array");
        return false;
    }//add

    public E search (E findMe){
        int i = 0;
        while (i < myCount && !(myArray[i].equals(findMe))){
            i++;
        }//while
        return (i < myCount ? myArray[i] : null);
    }//search

    private void swap (int firstPos, int otherPos){
        E temp = myArray[firstPos];
        myArray[firstPos] = myArray[otherPos];
```

```
        myArray[otherPos] = temp;
    }//swap

    public void bubbleSort(){
        for (int passNum = 1; passNum < myCount; passNum++) {
            for(int j = 0; j < myCount - passNum; j++) {
                if(myArray[j + 1].compareTo( myArray[j]) < 0 ){
                    swap(j, j + 1);
                }//if
            }//inner for
        }//outer for
    }//bubbleSort

    public String toString(){
      String stringToReturn = "";
        for (int i = 0; i < myCount; i++){
          stringToReturn += myArray[i] + "\n";//"\n" for the new line character
        }//for
        return stringToReturn;
    }//toString

}// List<E extends Comparable<E>>
```

Declaring and Constructing Comparator Objects

It is natural to think of names being put into alphabetical order. The `compareTo` method of a class in Java is said to represent the *natural ordering* of objects of that class. Some classes of objects may have alternative ordering schemes beyond the natural ordering. In the package `java.util`, the Java API provides the programmer with the `Comparator<E>` interface as a means of representing additional schemes for ordering objects of a class `E`. Classes that implement the `Comparator<E>` interface are used to construct objects whose only reason for existence is to have a method named `compare` that is passed two objects of class `E` as parameters and returns an `int` value. As you may already have guessed, that `int` will be negative if the first parameter is before the second parameter, zero if the first and second parameters are considered equal, and positive if the first parameter is after the second parameter, all with respect to the secondary ordering scheme that the class represents. Thus, in order to implement the `Comparator<E>` interface, a class must provide the details of a method with the following return type and signature:

```
int compare(E firstE, E secondE)
```

Suppose, for example, that we occasionally need an alternative ordering scheme for our `Name` class that put names into *reverse alphabetical order*. We can write a class that implements `Comparator<Name>` fairly easily. The `compare` method is similar to the `compareTo` method we have already written except that we have two parameter `Name`s instead of an invoking `Name` and one parameter `Name`. Also, the `compare` method needs to return the

arithmetic opposite of what the compareTo method returns. We accomplish this with a single minus as follows:

```
public int compare(Name name1, Name name2){
            return –name1.compareTo(name2);
}//compare
```

A question arises about where this class should be declared. Somehow, the Name class seems like a logical home for such a class that will work with Name objects.

Java allows for several kinds of classes declared within other classes. We have seen *inner classes* in previous chapters that implement the ActionListener interface. Each object constructed from an inner class has an automatic link to an object of the enclosing class that was the context for the inner class object's construction. It made sense in that context to use an inner class because a handler object was really linked to the application object that constructed it. In the case of Comparator<E> objects, there is no individual instance of class E that is appropriate to link to the comparator object who will be used to compare arbitrary pairs of objects of class E. In Java, classes declared as static within other classes are independent of any particular instance of the enclosing class. Hence, it is appropriate to declare a class that implements the Comparator<Name> interface as a static class within the Name class. Hence, we will use the key word static to the header of class ReverseAlphaComp.

The next question is: which class or object should construct an object of class ReverseAlphaComp? Once again, it makes sense that the Name class should construct such a comparator of its instances and make that comparator available to other classes that use objects of the Name class. It should also be the case that once a comparator is constructed, it should not be possible for another class to change it. To summarize, within the Name class we will construct an object of class ReverseAlphaComp and declare a public static final reference named REVERSE_COMPARATOR that will refer to it. Finally to keep the Name class in complete control of this comparator class, we declare the class ReverseAlphaComp itself as private. We have assembled the revised Name class below and have highlighted the new code. Note that we import java.util.* because that's where the Comparator<E> interface is located in the hierarchy.

```
import java.util.*;

public class Name implements Comparable<Name>{

    public static final ReverseAlphaComp
                    REVERSE_COMPARATOR = new ReverseAlphaComp();

    private String myFirst; // reference to the first name
    private String myMiddle; // reference to the middle name
    private String myFamily; // reference to the family name

    public Name (String theFirst, String theMiddle, String theFamily) {
           myFirst = theFirst;
           myMiddle = theMiddle;
           myFamily = theFamily;
```

```
}//3 parameter constructor

public Name (String theFirst, String theFamily) {
        myFirst = theFirst;
        myMiddle = "";
        myFamily = theFamily;
}//2 parameter constructor

public String getFirst() {
        return myFirst;
}//getFirst

public void setFirst(String theFirst) {
        myFirst = theFirst;
}//setFirst

public String getMiddle() {
        return myMiddle;
}//getMiddle

public void setMiddle(String theMiddle) {
        myMiddle = theMiddle;
}//setMiddle

public String getFamily() {
        return myFamily;
}//getFamily

public void setFamily(String theFamily) {
        myFamily = theFamily;
}//setFamily

public boolean equals(Object theOther){
        Name theName = (Name)theOther;
        return myFirst.equals(theName.myFirst)
                && myMiddle.equals(theName.myMiddle)
                && myFamily.equals(theName.myFamily);
}//equals

public int compareTo(Name theName){
        int compareFamily = myFamily.compareTo(theName.myFamily);
        if(compareFamily != 0) {
                return compareFamily;
        }//family names were different

        int compareFirst = myFirst.compareTo(theName.myFirst);
        if(compareFirst != 0) {
                return compareFirst;
        }//family names were equal and first names were different

        return myMiddle.compareTo(theName.myMiddle);
```

```
    }//compareTo

    public String toString() {
           return myFamily + ", " + myFirst +
                  (myMiddle.equals("") ? "" : " " + myMiddle);
    }//toString

    private static class ReverseAlphaComp implements Comparator<Name>{
           public int compare(Name name1, Name name2){
                  return –name1.compareTo(name2);
           }//compare
    }//ReverseAlphaComp
}//Name
```

PREVIEW OF THE LAMBDA EXPRESSIONS OF JAVA 1.8

The code of the `compare` method above is a single line. It seems like we wrote four lines of code surrounding that one important line. If the code within the `compare` method was more complicated those other 4 lines would not seem so "heavy weight." Lambda expressions were added to version 1.8 of Java language in part to address this "heavy" nature of such code. If the compiler can detect that a `Comparator<Name>` object is to be constructed, it can determine from the code of the `Comparator<E>` interface that the only method that needs to be written is a `compare` method that has return type `int` and has two `Name` parameters. An interface requiring that only one method be written is a *functional interface* in Java. A lambda expression is a way to express the method that needs to be written in a very short format. The format is so short that `Comparator<Name>` object can be declared and constructed in just one line of code. In writing the code in this manner the class that implements `Comparator<Name>` will be anonymous. Hence there will be no need for a class named `ReverseAlphaComp`, thereby shortening the code.

The lambda expression has several possible forms. The most verbose has the parameters of the method to the left of the arrow token `->` and a block of code to the right of the arrow that is to be executed. If there is only one line of code in the block, the curly brackets can be omitted. Hence one form of the lambda expression that embodies the compare method above is

```
(Name name1, Name name2) -> -name1.compareTo(name2)
```

We declare `REVERSE_COMPARATOR` to be a `public static final` variable of class `Comparator<Name>` and use this expression on the right side of the assignment operator as follows:

```
public static final Comparator<Name>
       REVERSE_COMPARATOR = (Name name1, Name name2) -> -name1.compareTo(name2);
```

We can shorten the code even further because the compiler can infer the types of the parameters, again because it knows that a `Comparator<Name>` object must have a `compare` method that takes two `Name` parameters. We can simply leave out the declarations for the parameters as follows:

```
    public static final Comparator<Name>
            REVERSE_COMPARATOR = (name1, name2) -> -name1.compareTo(name2);
```

We assemble the Name class once again for reference below.

```
import java.util.*;

public class Name implements Comparable<Name>{

        public static final ReverseAlphaComp
                REVERSE_COMPARATOR = (name1, name2) -> -name1.compareTo(name2);

        private String myFirst; // reference to the first name
        private String myMiddle; // reference to the middle name
        private String myFamily; // reference to the family name
        public Name (String theFirst, String theMiddle, String theFamily) {

                myFirst = theFirst;
                myMiddle = theMiddle;
                myFamily = theFamily;

        }//3 parameter constructor

        public Name (String theFirst, String theFamily) {
                myFirst = theFirst;
                myMiddle = "";
                myFamily = theFamily;
        }//2 parameter constructor

        public String getFirst() {
                return myFirst;
        }//getFirst

        public void setFirst(String theFirst) {
                myFirst = theFirst;
        }//setFirst

        public String getMiddle() {
                return myMiddle;
        }//getMiddle

        public void setMiddle(String theMiddle) {
                myMiddle = theMiddle;
        }//setMiddle

        public String getFamily() {
                return myFamily;
        }//getFamily

        public void setFamily(String theFamily) {
                myFamily = theFamily;
        }//setFamily

        public boolean equals(Object theOther){
                Name theName = (Name)theOther;
```

```
            return myFirst.equals(theName.myFirst)
                    && myMiddle.equals(theName.myMiddle)
                    && myFamily.equals(theName.myFamily);
    }//equals

    public int compareTo(Name theName){
            int compareFamily = myFamily.compareTo(theName.myFamily);
            if(compareFamily != 0) {
                    return compareFamily;
            }//family names were different

            int compareFirst = myFirst.compareTo(theName.myFirst);
            if(compareFirst != 0) {
                    return compareFirst;
            }//family names were equal and first names were different

            return myMiddle.compareTo(theName.myMiddle);

    }//compareTo

    public String toString() {
            return myFamily + ", " + myFirst +
                    (myMiddle.equals("") ? "" : " " + myMiddle);
    }//toString
}//Name
```

Using Comparator Objects

The next question to answer is how can our class `List<E>` be revised so that it can *either* construct a list object that is ordered by the natural ordering of its elements *or* construct a list object that is ordered by a comparator object for its elements? Here we acknowledge code written by Mark Allen Weiss for our strategy.

We add a `Comparator<E>` instance variable to class. In addition, we write a new pair of constructors that each take a `Comparator<E>` parameter. These new constructors do the same assignments that the original constructors perform. In addition, if one of these new constructors is invoked, the `Comparator<E>` instance variable is assigned the `Comparator<E>` parameter. We add a line of code to the old constructors that assigns `null` to the `Comparator<E>` instance variable by default.

We also add a new `private` method named `myCompare`. `myCompare` will select between using the natural ordering supplied through the `compareTo` method or the alternative ordering supplied through the `Comparator<E>` instance variable and its `compare` method.

`myCompare` takes two parameters of class `E` and returns an `int`. If the `Comparator<E>` instance variable is `null`, `myCompare` uses its first parameter as the invoking object for the `compareTo` method, the second parameter as the argument for `compareTo`, and returns the `int` value that `compareTo` returns to it. If the `Comparator<E>` instance variable is not `null`, `myCompare` invokes the `compare` method on the `Comparator<E>` instance variable with the two parameters of `myCompare` as its arguments and returns the `int` value that `compare` returns to it. The new code is highlighted below.

```
import java.util.*;// for the Comparator<E> interface
```

```
public class List <E extends Comparable<E>> {

    private E myArray[]; // array of Objects
    private int myCount;// current count of Objects in List
    private Comparator<E> myComparator;// comparator for alternate ordering scheme

    public List () {
         myArray = (E []) new Comparable[100];
         myCount = 0;
         myComparator = null;
    }//0 parameter constructor

    public List (int theSize) {
         myArray = (E []) new Comparable[theSize];
         myCount = 0;
         myComparator = null;
    }//1 parameter constructor

    public List (Comparator<E> theComparator) {
           myArray = (E []) new Comparable[100];
           myCount = 0;
           myComparator = theComparator;
    }//1 parameter constructor

    public List (int theSize, Comparator<E> theComparator) {
           myArray = (E []) new Comparable[theSize];
           myCount = 0;
           myComparator = theComparator;
    }//2 parameter constructor

    private int myCompare (E firstE, E secondE){
           if(myComparator == null) {
                  return firstE.compareTo(secondE);
           }//no comparator
           return myComparator.compare(firstE, secondE);
    }//myCompare
```

Finally, we change the comparison line in `bubbleSort` so that it invokes `myCompare` with the two references to array elements as arguments as follows:

```
    public void bubbleSort(){
         for (int passNum = 1; passNum < myCount; passNum++) {
               for(int j = 0; j < myCount - passNum; j++) {
                     if(myCompare(myArray[j + 1],myArray[j]) < 0 ){
                           swap(j, j + 1);
                     }//if
               }//inner for
         }//outer for
    }//bubbleSort
```

A test program showing construction of one list ordered by the natural ordering and another ordered by the comparator is displayed below.

```
public class TestNameList {
  public static void main(String[] args) {
    List<Name> myList = new List<Name>();
    List<Name> myListR = new List<Name>(Name.REVERSE_COMPARATOR);
    Name name1 = new Name("Elizabeth", "Catherine", "Li Santi");
    Name name2 = new Name("Lydia", "Michelle", "Mann");
    Name name3 = new Name("Barbara", "Ann-Teresa", "Li Santi");
    if(myList.add(name1)){
      System.out.println("Elizabeth Catherine Li Santi added successfully" +
              " to first list.");
    }else {
      System.out.println("Elizabeth Catherine Li Santi not added to first list.");
    }//else
    if(myList.add(name2)){
      System.out.println("Lydia Michelle Mann added successfully" +
            " to first list.");
    }else {
      System.out.println("Lydia Michelle Mann not added to first list.");
    }//else
    if(myList.add(name3)){
      System.out.println("Barbara Ann-Teresa Li Santi added successfully" +
            " to first list.");
    }else {
      System.out.println("Barbara Ann-Teresa Li Santi not added to first list.");
    }//else
    if(myListR.add(name1)){
      System.out.println("Elizabeth Catherine Li Santi added successfully" +
            " to second list.");
    }else {
      System.out.println("Elizabeth Catherine Li Santi not added to second list.");
    }//else
    if(myListR.add(name2)){
      System.out.println("Lydia Michelle Mann added successfully" +
            " to second list.");
    }else {
      System.out.println("Lydia Michelle Mann not added to second list.");
    }//else
    if(myListR.add(name3)){
      System.out.println("Barbara Ann-Teresa Li Santi added successfully" +
            " to second list.");
    }else {
      System.out.println("Barbara Ann-Teresa Li Santi not added to second list.");
```

```
    }//else
    System.out.println("The first list is:\n" + myList.toString() + "\n");
    myList.bubbleSort();
    System.out.println("The first sorted list is:\n"+myList.toString() + "\n");
    System.out.println("The second list is:\n" + myListR.toString() + "\n");
    myListR.bubbleSort();
    System.out.println("The second sorted list is:\n"+myListR.toString()+"\n");
  }//main

}//TestNameList
```

The Diamond Operator, <>, of Java 1.7

Programmers quickly noted the redundancy in lines of code like those above:

```
List<Name> myList = new List<Name>();
List<Name> myListR = new List<Name>(Name.REVERSE_COMPARATOR);
```

Using inheritance, it is possible that the type parameter of the declared type appearing on the left of each assignment statement might be different from the type parameter of the actual type being constructed on the right side of each assignment statement. However, in many circumstances, those two type parameters are exactly the same. Programmers felt they did not need to specify the type parameter twice. Hence, in Java 1.7, a new operator called the "diamond" operator and denoted <>, was introduced. The diamond operator allows the type parameter to be omitted on the right side of the assignment operator when it is the same as the type parameter already used in the declaration of the variable. The diamond operator determines the appropriate type from the type parameter used in the declaration. Hence, the lines above can be shortened if the Java compiler being used is at least 1.7 to the following:

```
List<Name> myList = new List<>();
List<Name> myListR = new List<>(Name.REVERSE_COMPARATOR);
```

Writing a Generic Method for Bubble Sort

We have written the `bubbleSort` method and its supporting `swap` method in the context of our class `List<E>` as follows:

```
public void bubbleSort(){
    for (int passNum = 1; passNum < myCount; passNum++) {
        for(int j = 0; j < myCount - passNum; j++) {
            if(myCompare(myArray[j + 1], myArray[j]) < 0 ){
                swap(j, j + 1);
            }//if
        }//inner for
    }//outer for
```

```
}//bubbleSort

private void swap (int firstPos, int otherPos){
    E temp = myArray[firstPos];
    myArray[firstPos] = myArray[otherPos];
    myArray[otherPos] = temp;
}//swap
```

Our bubbleSort and swap methods were declared within a class where the identifiers for myArray and myCount are within scope. In other words, our bubbleSort and swap methods did not need to be passed those variables as parameters. In addition, the type identifier E was known to the compiler from the class header.

The bubbleSort method can be written as a utility method in a more independent way in the context of a class that provides sort methods to other classes. The bubbleSort method will be a *generic* static method in the utility class. The utility class will not have the reference myArray or know the count of elements in myArray. Hence, the reference to myArray and the count of elements in myArray must be passed as arguments when bubbleSort is invoked. The compiler will also need a way to make sense of the type parameter E, that describes the kind of references in the array if the compareTo method is to be invoked on the array elements. In the case of a secondary ordering provided by a Comparator<E> object, a second bubbleSort method will be written that requires a reference to that Comparator<E> object also be passed as an additional parameter. Our List<E> class will select the appropriate bubbleSort method dependent on whether the myComparator instance variable is null or not.

We will also need to rewrite the private swap method invoked by bubbleSort in order to make the generic bubbleSort work since it also uses myArray. It should accept a parameter that is a reference to the array in addition to the ints for the two positions of the references to be swapped. Additionally, because the generic bubbleSort method is static, and the swap method is in the same class, the swap method must also be declared as static. In Java, if aMethod and bMethod are in class C and aMethod is static, aMethod can only call bMethod if bMethod is also static.

Below in the rewritten bubbleSort and swap methods, we will use the identifier theArray for the array as a parameter, and theCount for the int parameter for the count of elements in the array. In the case of the second bubbleSort method, we will use the identifier theComparator for the Comparator<E> parameter passed to the method.

The syntax of the method header for the first bubbleSort that relies on a compareTo method is a bit complicated. It is between static and the return type void that we give the compiler the information it needs about the type parameter E. As in our previous code, <E extends Comparable<E>>.

The header for the swap method is not as complicated because swap does not need to know that the objects are comparable. We simply indicate the type parameter <E> between static and the return type void so that the compiler knows that the identifier E is a type parameter. Note, that like bubbleSort, we have a parameter that is a reference to an array of E.

The first bubbleSort method and the swap method are below. Note that swap is called with three arguments.

```
public static <E extends Comparable<E>> void bubbleSort(E[] theArray, int theCount){
      for (int passNum = 1; passNum < theCount; passNum++) {
         for(int j = 0; j < theCount – passNum; j++) {
            if(theArray[j + 1].compareTo( theArray[j]) < 0 ){
               swap(theArray, j, j + 1);
            }//if
         }//inner for
      }//outer for
}//bubbleSort
private static <E> void swap (E[] theArray, int firstPos, int secondPos){
      E temp = theArray[firstPos];
      theArray[firstPos] = theArray[secondPos];
      theArray[secondPos] = temp;
}//swap
```

The syntax of the method header for the second `bubbleSort` that does not rely on a `compareTo` method is simpler in that respect, but it has a third parameter for the `Comparator<E>` reference. It is displayed below. Note the use of the `compare` method of the `Comparator<E>` object instead of the `compareTo` method.

```
public static <E> void bubbleSort
               (E[] theArray, int theCount, Comparator<E> theComparator){
      for (int passNum = 1; passNum < theCount; passNum++) {
         for(int j = 0; j < theCount – passNum; j++) {
            if(theComparator.compare(theArray[j + 1],theArray[j]) < 0 ){
               swap(theArray, j, j + 1);
            }//if
      }//inner for
   }//outer for
}//bubbleSort with Comparator
```

We can now write a new class named `SortingMethods` and place the `bubbleSort` and `swap` methods in that class as follows:

```
public class SortingMethods {
      private static <E> void swap (E[] theArray, int firstPos, int secondPos){
         E temp = theArray[firstPos];
         theArray[firstPos] = theArray[secondPos];
         theArray[secondPos] = temp;
      }//swap
      public static <E extends Comparable<E>> void bubbleSort
                     (E[] theArray, int theCount){
         for (int passNum = 1; passNum < theCount; passNum++) {
            for(int j = 0; j < theCount – passNum; j++) {
               if(theArray[j + 1].compareTo( theArray[j]) < 0 ){
                  swap(theArray, j, j + 1);
               }//if
```

```
        }//inner for
      }//outer for
   }//bubbleSort without comparator
   public static <E> void
      bubbleSort(E[] theArray, int theCount, Comparator<E> theComparator){
         for (int passNum = 1; passNum < theCount; passNum++) {
            for(int j = 0; j < theCount – passNum; j++) {
               if(theComparator.compare(theArray[j + 1],theArray[j]) < 0 ){
                  swap(theArray, j, j + 1);
               }//if
            }//inner for
         }//outer for
   }//bubbleSort with Comparator

}//SortingMethods
```

Finally, we replace the `bubbleSort` and `swap` methods of our `List<E extends Comparable<E>>` class with a single `bubbleSort` method that selects the appropriate generic `bubbleSort` method of class `SortingMethods` to invoke. Note that we have *overloaded* the `bubbleSort` method by having two versions with different parameter lists. When we send the appropriate instance variables to `SortingMethods.bubbleSort`, the compiler chooses the correct version of the method.

```
   public void bubbleSort(){
        if (myComparator == null) {
              SortingMethods.bubbleSort(myArray, myCount);
        } else {
              SortingMethods.bubbleSort(myArray, myCount, myComparator);
        }//else
   }//bubbleSort
```

A small test program for our generic `bubbleSort` methods is shown below.

Our class `TestSortingMethods` has a generic `display` method so that we can view the results of our sort.

```
public class TestSortingMethods {

  public static void main(String[] args) {
    List<Name> myList = new List<>();
    List<Name> myListR = new List<>(Name.REVERSE_COMPARATOR);
    Name name1 = new Name("Elizabeth", "Catherine", "Li Santi");
    Name name2 = new Name("Lydia", "Michelle", "Mann");
    Name name3 = new Name("Barbara", "Ann–Teresa", "Li Santi");
    if(myList.add(name1)){
      System.out.println("Elizabeth Catherine Li Santi added successfully" +
          " to first list.");
    }else {
      System.out.println("Elizabeth Catherine Li Santi not added to first list.");
```

```
    }//else
    if(myList.add(name2)){
      System.out.println("Lydia Michelle Mann added successfully" +
            " to first list.");
    }else {
      System.out.println("Lydia Michelle Mann not added to first list.");
    }//else
    if(myList.add(name3)){
      System.out.println("Barbara Ann–Teresa Li Santi added successfully" +
          " to first list.");
    }else {
      System.out.println("Barbara Ann–Teresa Li Santi not added to first list.");
    }//else
    if(myListR.add(name1)){
      System.out.println("Elizabeth Catherine Li Santi added successfully" +
            " to second list.");
    }else {
      System.out.println("Elizabeth Catherine Li Santi not added to second list.");
    }//else
    if(myListR.add(name2)){
      System.out.println("Lydia Michelle Mann added successfully" +
            " to second list.");
    }else {
      System.out.println("Lydia Michelle Mann not added to second list.");
    }//else
    if(myListR.add(name3)){
      System.out.println("Barbara Ann–Teresa Li Santi added successfully"+
            " to second list.");
    }else {
      System.out.println("Barbara Ann–Teresa Li Santi not added to second list.");
    }//else
    System.out.println("\nThe first list is:\n" + myList + "\n");
    myList.bubbleSort();
    System.out.println("The first sorted list is:\n"+myList + "\n");
    System.out.println("The second list is:\n" + myListR + "\n");
    myListR .bubbleSort();
    System.out.println("The second sorted list is:\n"+myListR + "\n");
  }//main
}// TestSortingMethods
```

The output from the test program is as follows. Note that the second list is sorted into reverse alphabetical order.

```
Elizabeth Catherine Li Santi added successfully to first list.
Lydia Michelle Mann added successfully to first list.
Barbara Ann-Teresa Li Santi added successfully to first list.
Elizabeth Catherine Li Santi added successfully to second list.
Lydia Michelle Mann added successfully to second list.
Barbara Ann-Teresa Li Santi added successfully to second list.

The first list is:
Li Santi, Elizabeth Catherine
Mann, Lydia Michelle
Li Santi, Barbara Ann-Teresa

The first sorted list is:
Li Santi, Barbara Ann-Teresa
Li Santi, Elizabeth Catherine
Mann, Lydia Michelle

The second list is:
Li Santi, Elizabeth Catherine
Mann, Lydia Michelle
Li Santi, Barbara Ann-Teresa

The second sorted list is:
Mann, Lydia Michelle
Li Santi, Elizabeth Catherine
Li Santi, Barbara Ann-Teresa
```

Exercise 12.2:

1) Write two additional methods in class `SortingMethods` that implement `selectionSort` for `Lists` of `Comparable` objects and for `Lists` that employ a `Comparator` object.
2) Write a `selectionSort` method in class `List<E extends Comparable<E>>` that selects and invokes the appropriate `selectionSort` method of class `SortingMethods` to sort the `List`.
3) Add code to the class `TestSortingMethods` to test the methods you wrote for parts 1) and 2) above.

Exercise 12.3:

1) Write two additional methods in class `SortingMethods` that implement `insertionSort` for `Lists` of `Comparable` objects and for `Lists` that employ a `Comparator` object.
2) Write an `insertionSort` method in class `List<E extends Comparable<E>>` that selects and invokes the appropriate `selectionSort` method of class `SortingMethods` to sort the `List`.
3) Add code to the class `TestSortingMethods` to test the methods you wrote for parts 1) and 2) above.

Exercise 12.4: Continuation of Exercise 11.10

1) Add features to your class `AddressInfo` from Chapter 11:
   a) Declare a `private static class AddressComp` that `implements Comparator<AddressInfo>` within class `AddressInfo`. The `compare` method of class `AddressComp` will be used to implement the scheme of ordering `AddressInfo` objects by address.
   b) Also within class `AddressInfo` declare a `public static final` reference named `ADDRESS_COMPARATOR` to an object of class `AddressComp`, construct an object of class `AddressComp`, and assign the object to `ADDRESS_COMPARATOR`.

2) Replace your class `AddressInfoList` with a class `List` that has a type parameter `E` that extends `Comparable<E>` and is very similar to the `List<E extends Comparable<E>>` class in this chapter. (Choose **Rename...** from Eclipse's **Refactor** menu to rename your `AddressInfoList` class as the `List` class and then make the following modifications.)
   a) Your new `List<E extends Comparable<E>>` class will have an instance variable that is an array of references to objects of type `E` instead of an array of references to objects of type `AddressInfo`.
   b) Your new `List<E extends Comparable<E>>` class should have an additional instance variable with declared type `Comparator<E>`.
   c) Modify your existing constructor so that it constructs an array of `Comparable`'s that is cast to `E[]` and set the `Comparator<E>` instance variable to `null`.
   d) Write one more constructor that has an additional parameter that is a `Comparator<E>` reference and assign it to the `Comparator<E>` instance variable.
   e) Write a `myCompare` method like the `myCompare` method of class `List<E extends Comparable<E>>` in this chapter that selects between use of the `compareTo` method of its first parameter or uses the `compare` method of its `Comparator<E>` instance variable.
   f) Replace your two sort methods with one `sort` method that uses the `myCompare` method.
   g) Replace any remaining occurrences of `AddressInfo` to the generic type variable `E`.

3) Revise your `Database` class so that you declare, construct, and use two `List<E extends Comparable<E>>` objects, one that can be sorted alphabetically and one that can be sorted by address. More specifically, be sure that:
   a) When you construct your list sorted by address that you pass the public reference to the `AddressComp` object to the constructor.
   b) When you add a new record to the database, you add it to both lists.
   c) When you delete a record from the database, you delete it from both lists.
   d) Your alphabetical display button needs only sort and display the alphabetical list.
   e) Your address display button needs only sort and display the list that is sorted by address.

Exercise 12.5: Continuation of Exercise 11.12

1) Add features to your class `BirthInfo` from Chapter 11.
   a) Declare a `private static class ChronComp` that `implements Comparator<BirthInfo>` within class `BirthInfo`. The `compare` method of class `ChronComp` will be used to implement the scheme of ordering `BirthInfo` objects by birthday.
   b) Also within class `BirthInfo` declare a `public static final` reference named `DATE_COMPARATOR` to an object of class `ChronComp`, construct an object of class `ChronComp`, and assign the object to `DATE_COMPARATOR`.
2) Replace your class `BirthInfoList` with a class `List` that has a type parameter `E` that extends `Comparable<E>` and is very similar to the `List<E extends Comparable<E>>` class. (Choose **Rename...** from Eclipse's **Refactor** menu to rename your `BirthInfoList` class as the `List` class and then make the following modifications.)
   a) Your new `List<E extends Comparable<E>>` class will have an instance variable that is an array of references to objects of type `E` instead of an array of references to objects of type `BirthInfo`.
   b) Your new `List<E extends Comparable<E>>` class should have an additional instance variable with declared type `Comparator<E>`.
   c) Modify your two existing constructors so that they construct an array of `Comparable`'s that is cast to `E[]` and set the `Comparator<E>` instance variable to `null`.
   d) Write two more constructors that take an additional parameter that is a `Comparator<E>` reference and assign it to the `Comparator<E>` instance variable.
   e) Write a `myCompare` method like the `myCompare` method of class `List<E extends Comparable<E>>` that selects between use of the `compareTo` method of its first parameter or uses the `compare` method of its `Comparator<E>` instance variable.
   f) Replace your two sort methods with one `sort` method that uses the `myCompare` method.
   g) Replace any remaining occurrences of `BirthInfo` to the generic type variable `E`.
3) Revise your `Database` class so that you declare, construct, and use two `List<BirthInfo>` objects, one that can be sorted alphabetically and one that can be sorted chronologically. More specifically, be sure that:
   a) When you construct your chronological list, you pass the public reference to the `ChronComp` object to the constructor.
   b) When your code adds a new record to the database, add the new record to both lists.
   c) When your code deletes a record from the database, delete the record from both lists.
   d) Your alphabetical display button needs only sort and display the alphabetical list.
   e) Your chronological display button needs only sort and display the chronological list.

Exercise 12.6: A stack is a last in first out (LIFO) data structure. This means that the last item placed on a stack is the first item to come off the stack. A good example is a stack of dishes in a cupboard. When you take a plate from the drying rack to put it away, you put it on the top of the stack of plates in the cupboard. When you need a plate on which to put your sandwich, the last plate you put on the stack in the cupboard is the first one you will take off the stack.

The following public methods are usually provided for a stack:

- A zero parameter constructor that constructs a stack of a fixed size.
- A one parameter constructor that constructs a stack of the specified size.
- A `push` method that has on `Object` as a parameter, attempts to push its parameter onto the top of the stack, and returns `true` if successful or `false` if not successful.
- A `pop` method that pops an `Object` off the top of the stack and returns a reference to that `Object`.
- An `isEmpty` method that returns `true` if the stack is empty and `false` otherwise.
- A `peek` method that returns a reference to the top object on the stack, so that you can "examine" it. The `peek` method does **not** remove the top object.
- A `toString` method that returns a `String` representation of the stack.

Implementing a stack: One way to implement a stack is to use an array. The 0th position in the array represents the bottom of the stack, so the stack grows from that position. An integer is used to keep track of the top of the stack.

Your goal is to write a class named `WCSStack<E>`.

1) Draw several pictures, showing that you understand the operation (methods) of a stack.
2) Write the complete class `WCSStack`. Your class should have a type parameter `E`.
3) Test your class using the GUI named `StackTest` that is displayed below. A `UtilityMethods` class is employed to construct the GUI components and provide window closing behavior. Both of the classes, `StackTest` and `UtilityMethods` are provided in the package named `for_stack`. Although your stack will hold generic objects, `StackTest` will test it by using `String` objects.

```
package for_stack;
/*
 * A GUI provided to test the WCSStack class
 */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

    public StackTest() {
        super("The Stack Tester");
        setLocation(100, 100);
        setSize(400, 450);
        myCP = getContentPane();
        myCP.setLayout(null);
```

(continues)

Exercise 12.6, continues:

```
        pushB = UtilityMethods.makeButton("PUSH", 75, 10, 80, 30, new PushBHandler(), myCP);
        popB = UtilityMethods.makeButton("POP", 175, 10, 80, 30, new PopBHandler(), myCP);
        peekB =UtilityMethods.makeButton("PEEK", 275, 10, 80, 30, new PeekBHandler(), myCP);

        inputL = UtilityMethods.makeLabel("Input", 100, 60, 200, 20, myCP);
        stackL = UtilityMethods.makeLabel("The Stack", 100, 310, 200, 30, myCP);

        inputTF = UtilityMethods.makeTextField("", 100, 80, 200, 30, myCP);
        messageTF = UtilityMethods.makeTextField("", 10, 370, 380, 30, myCP);

        stackTA = new JTextArea();
        stackSP = UtilityMethods.makeScrollPane(stackTA, 100, 120, 200, 200, myCP);

        UtilityMethods.closingCausesExit(this);

            setVisible(true);
            myStack = new WCSStack<String>();

        } //constructor StackTest

        public class PushBHandler implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                String userInput = inputTF.getText();
                if(userInput.equals("")){
                    messageTF.setText("Nothing to push onto the stack.");
                } else {
                    if (myStack.push(userInput)) {
                        messageTF.setText(userInput + " has been pushed onto the stack.");
                    } else {
                        messageTF.setText("Error in pushing " + userInput +
                        " onto the stack.");
                    }//else
                }//outer else
                stackTA.setText(myStack.toString());
                inputTF.setText("");
                inputTF.requestFocus();
            }//actionPerformed
        }//PushBHandler

        public class PopBHandler implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                String fromStack;
                if(!myStack.isEmpty() ) {
                    fromStack = myStack.pop();
                    messageTF.setText("The top item, " + fromStack +
                    " has been popped off the stack.");
                    stackTA.setText(myStack.toString());
                } else {
```

(continues)

Exercise 12.25, continues:

```
                messageTF.setText("The stack is empty. Nothing can be popped off.");
            }//else
            inputTF.requestFocus();
        }//actionPerformed
    }//PopBHandler

    public class PeekBHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if(!myStack.isEmpty()) {
                messageTF.setText(myStack.peek() + " is on the top of the stack.");
            } else {
                messageTF.setText("The Stack is empty.");
            }//else
            inputTF.requestFocus();
        }//actionPerformed
    }//PeekBHandler

    public static void main (String args[]) {
        StackTest myApp = new StackTest();
    } //main
} //StackTest
```

```
package for_stack;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class UtilityMethods {

    public static JButton makeButton(String text, int x, int y, int w, int h,
            ActionListener theListener, Container theCP){
        JButton buttonToReturn = new JButton(text);
        buttonToReturn.setLocation(x, y);
        buttonToReturn.setSize(w, h);
        buttonToReturn.addActionListener(theListener);
        theCP.add(buttonToReturn);
        return buttonToReturn;
    }//makeButton

    public static JLabel makeLabel(String text, int x, int y, int w, int h,
                          Container theCP){
        JLabel labelToReturn = new JLabel(text, JLabel.CENTER);
        labelToReturn.setLocation(x, y);
        labelToReturn.setSize(w, h);
        theCP.add(labelToReturn);
        return labelToReturn;
    }//makeLabel
```

(continues)

Exercise 12.6, continues:

```
   public static JScrollPane makeScrollPane(JTextArea theClient, int x, int y,
          int w, int h, Container theCP){
      JScrollPane scrollPaneToReturn = new JScrollPane(theClient,
           JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
           JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
      scrollPaneToReturn.setLocation(x, y);
      scrollPaneToReturn.setSize(w, h);
      theCP.add(scrollPaneToReturn);
      return scrollPaneToReturn;
   }//makeLabel

   public static JTextField makeTextField(String text, int x, int y, int w, int h,
            Container theCP){
      JTextField textFieldToReturn = new JTextField(text);
      textFieldToReturn.setLocation(x, y);
      textFieldToReturn.setSize(w, h);
      theCP.add(textFieldToReturn);
      return textFieldToReturn;
   }//makeTextField

   public static void closingCausesExit(JFrame theWindow) {
      theWindow.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
           System.exit(0);
         }//windowClosing
      }); //end of definition of WindowAdapter and semicolon to end the line
   }//closingCausesExit

}//UtilityMethods
```

Exercise 12.7: A queue is a first in first out (FIFO) data structure. This means that the first item to enter a queue is the first item to leave the queue. A good example is a line of cars waiting to pass through a toll gate. You get in line at the end of the line. You exit the line when you are at the head of the line. The first car to get out of the line (through the toll gate) is the first car that got into the line.

The following public methods are usually provided for a queue:

- A zero parameter constructor that constructs a queue of a fixed size.
- A one parameter constructor that constructs a queue of the specified size.
- An `enter` method that has an `Object` as a parameter, attempts to place its parameter at the end of the queue, and returns `true` if successful or `false` if not successful.
- A `dequeue` method that takes an `Object` off the head of the queue and returns a reference to that `Object`.
- An `isEmpty` method that returns `true` if the queue is empty and `false` otherwise.

(continues)

Exercise 12.7, continues:

- A peek method that returns a reference to the top object at the head of the queue, so that you can "examine" it.
- A toString method that returns a String representation of the queue.

Implementing a queue: One way to implement a stack is to use an array. The 0th position in the array represents the head of the queue, so the queue grows from that position. An integer is used to keep track of the end of the queue. When an object leaves the queue, all other objects in the queue move one position closer to the head.

Your goal is to write a class named WCSQueue<E>.

1) Draw several pictures, showing that you understand the operation (methods) of a queue.
2) Write the complete class. Your class should have a type parameter **E**.
3) Test your class using the GUI named QueueTest that is displayed below. A UtilityMethods class is employed to construct the GUI components and provide window closing behavior. The UtilityMethods class is displayed at the end of Exercise 12.6. Both of the classes, QueueTest and UtilityMethods are provided in the package named for_queue. Although your queue will hold generic objects, you will test it by using String objects.

```
package for_queue;

/*
 * A GUI provided to test a Queue class
 */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class QueueTest extends JFrame {

    private JButton enterQB, dequeueB, peekB;//buttons for the usual queue operations
    private JTextArea queueTA;//where the contents of the queue are displayed
    private JScrollPane queueSP;//controls the queue text area
    private JTextField inputTF, messageTF;//where user enters input for the queue
                                //and where messages to the user are displayed
    private JLabel inputL, queueL;//labels for the text fields
    private Container myCP;//reference to the content pane
    private WCSQueue<String> myQueue;//reference to the queue object whose methods
                            //must work with this class

    public QueueTest() {
        super("The Queue Tester");
        setLocation(100, 100);
        setSize(600, 450);
        myCP = getContentPane();
        myCP.setLayout(null);
```

(continues)

Exercise 12.7, continues:

```
            enterQB = UtilityMethods.makeButton("Enter Queue", 50, 10, 150, 30,
                        new EnterQBHandler(), myCP);
            dequeueB = UtilityMethods.makeButton("Dequeue", 225, 10, 150, 30,
                        new DequeueBHandler(), myCP);
            peekB = UtilityMethods.makeButton("Peek", 400, 10, 150, 30,
                        new PeekBHandler(), myCP);

            inputL = UtilityMethods.makeLabel("Input", 200, 60, 200, 20, myCP);
            queueL = UtilityMethods.makeLabel("The Queue", 200, 310, 200, 30, myCP);

            inputTF = UtilityMethods.makeTextField("", 200, 80, 200, 30, myCP);
            messageTF = UtilityMethods.makeTextField("", 50, 370, 500, 30, myCP);

            queueTA = new JTextArea();
            queueSP = UtilityMethods.makeScrollPane(queueTA,100, 120, 400, 200, myCP);

            setVisible(true);
            myQueue = new WCSQueue<String>();

        } //constructor QueueTest

        public class EnterQBHandler implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                String userInput = inputTF.getText();
                if(userInput.equals("")){
                    messageTF.setText("Nothing to enter the queue.");
                } else {
                    if (myQueue.enter(userInput)) {
                        messageTF.setText(userInput + " has entered the queue.");
                    } else {
                        messageTF.setText("Error in " + userInput +
                            " trying to enter the queue.");
                    }//else
                }//outer else
                queueTA.setText(myQueue.toString());
                inputTF.setText("");
                inputTF.requestFocus();
            }//actionPerformed
        }//EnterQBHandler

        public class DequeueBHandler implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                String fromQueue;
                if(!myQueue.isEmpty() ) {
                    fromQueue = myQueue.dequeue();
                    messageTF.setText("The item at the head of the queue, " + fromQueue +
                        " has left the queue.");
                    queueTA.setText(myQueue.toString());
```

(continues)

Exercise 12.7, continues:

```
            } else {
                messageTF.setText("The queue is empty.");
            }//else
            inputTF.requestFocus();
        }//actionPerformed
    }//DequeueBHandler
    public class PeekBHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if(!myQueue.isEmpty()) {
                messageTF.setText(myQueue.peek() + " is at the head of the queue.");
            } else {
                messageTF.setText("The Queue is empty.");
            }//else
            inputTF.requestFocus();
        }//actionPerformed
    }//PeekBHandler

    public static void main (String args[]) {
        QueueTest myApp = new QueueTest();
    } //main
} //QueueTest
```

Exercise 12.8: Revise your `WCSStack` class so that if there is an attempt to push an item onto the stack when the array is full, a new temporary array of double length is allocated, the references from the original array are copied into the temporary array, the reference to the original array is assigned to point to temporary array, and the application pushes the item onto the stack.

Exercise 12.9: Revise your `WCSQueue` class so that the queue "wraps around" the array. In other words, do not move references to lower index positions whenever an item leaves the queue. Instead, have instance variables that keep track of the position of the head of the queue, the position of the tail of the queue, and the number of items currently in the queue. In this way, the array should not be full unless the number of items currently in the array is equal to the length of the array. In addition, when the array is full and there is an attempt to enqueue another item, a new temporary array of double length is allocated, the references from the original array are copied into the temporary array with the item at the head of the original queue copied into the index `0` position in the temporary array, the reference to the original array is assigned to point to temporary array, and the application enqueues the item.

Lab: Generic Methods • Chapter 12

The objectives of this lab is:

- to get experience writing generic methods.

This lab uses Java classes named `GenericMethods` and `TestGenericMethods` in a code package entitled `for_generic_methods_lab`. The code you have been provided in the `main` method of class `TestGenericMethods` generates 100 random Integer values in the range 0–99 and stores them in an array. The code also stores ten `Strings` in an array of length 20. The code in class `GenericMethods` also includes a generic method that will implement bubble sort on an array with ordering based on the natural ordering of elements in the array.

For each of the tasks in the numbered steps described below, your goal is:

- to write a generic method in class `GenericMethods` that will work with either one of the arrays named `myIntegers` and `myStrings`
- to write code to test your method in the `main` method of class `TestGenericMethods`

The methods should work with a type parameter `E`. You should indicate that `E` extends `Comparable<E>` whenever necessary. If a method returns one of the objects in the array passed to it, then the return type of the method should be `E`. Please note the following specifications for each method you write:

- Within your code in class `GenericMethods`, please write a comment before each method giving the step number as indicated below.
- Write comments in the `main` method in class `TestGenericMethods` indicating the lines of code needed to accomplish testing each method.
- You should use additional variables and syntactic constructs, e.g., selection constructs to select between alternative literal strings, as needed in conjunction with `System.out.println` to show that your methods work correctly.
- To make your name appear on the output, use `System.out.println` in the `main` method to display your name before any other the output.
- On the hard copy of your output, annotate your output by hand to indicate which parts of the output correspond to tests for each method.
- Present methods in the `GenericMethods.java` file in order according to step number below.
- Make sure you display the contents of any array that has been modified.

Note: A sample solution is already provided in the code for the task of writing a method that determines and returns the maximum value stored in the array parameter.

1) Get a copy of the folder `for_generic_methods_lab` onto the **Desktop**.
2) 2) Launch Eclipse.
3) 3) Create a new Java project named with `GenericMethods` **and** `your name`.

4) Drag the folder for_generic_methods_lab from the **Desktop** onto the image of the your GenericMethods folder in the Package Explorer view.

5) Double click on your GenericMethods folder in the Package Explorer view to open the folder. You should see that the package included in your folder is named for_generic_methods_lab.

6) Double click on the for_generic_methods_lab package in the Package Explorer view to open the folder. You should see that the files named GenericMethods.java and TestGenericMethods.java is included in your package named for_generic_methods_lab.

7) Double click on GenericMethods.java to open the file in the editor view.

8) Double click on TestGenericMethods.java to open the file in the editor view.

9) Run the program.

10) Write and test a generic method that determines and returns the minimum value stored in the array parameter. Return null if there are no elements stored in the array.

11) Write and test a generic method that displays a list of all the elements of the array parameter that are greater than an E parameter.

12) Write and test a generic method that displays in reverse order the elements of the array parameter. This method should not change the positions of the elements within the array. It should just display the elements in the order that is the reverse of how they are stored in the array, i.e., from index theCount –1 to index 0.

13) Write and test a generic method that displays every other element of the array parameter, i.e., elements in indices 0, 2, 4, 6, ….

14) Write and test a generic method that determines and returns the index that is the first occurrence (start searching at index 0) of an E parameter in the array parameter. Return -1 if the parameter is not found in the array.

15) Write and test a generic method that determines and returns the index of the first occurrence (start searching at index 0) of a value in the array parameter that is greater than an E parameter. Return -1 if no element of the array is greater than the parameter.

16) Write and test a generic method that determines and returns the index that is the last occurrence (start searching at index theCount -1) of an E parameter in the array parameter. Return -1 if the parameter is not found in the array.

17) Write and test a generic method that determines and returns the index of the last occurrence (start searching at index theCount -1) of a value in the array parameter that is greater than an E parameter. Return -1 if no element of the array is greater than the parameter.

18) Write and test a generic method that replaces every value in the array parameter that equals an E parameter with the value null.

19) Write and test a generic method that reverses the order in which the elements are stored in the array parameter, i.e., the reference at index theCount -1 moves to index 0, the reference at index theCount -2 moves to index 1, etc.

20) Write and test a generic method that rotates the elements in the array parameter by one position. In other words, each reference is moved to the next higher index position, except the reference in position theCount -1, which should be moved to position 0.

21) Write and test a generic method that rotates the elements in the array parameter by two positions. In other words, each reference is moved two index positions higher, except the references in positions `theCount -2` and `theCount -1`, which should be moved into positions `0` and `1`, respectively.

22) Write and test a generic method that rotates the elements in the array parameter by an `int` parameter number of positions.

23) Write and test a generic method that displays all elements whose references are stored in the array parameter in indices that are multiples of 5 (i.e., indices `0`, `5`, `10`, `15`, `20`, `25`, ...)

24) Write and test a generic method that displays all elements whose references are stored in the array parameter in indices that are multiples of 4 (i.e., indices `0`, `4`, `8`, `12`, `16`, `20`, ...)

25) Write and test a generic method that displays all elements whose references are stored in the array parameter in indices that are multiples of an `int` parameter (i.e., if the parameter is `x`, then the indices are `0`, `2x`, `3x`, `4x`, ...)

26) Assuming that the array parameter has **not** been sorted, write and test a generic method that determines and returns the first reference stored in the array that points to an element that is not in ascending order. Since the method written for Step 17 may have replaced some references stored in the array with the value null, your method should skip over `null` if it is encountered. The JVM will complain if you attempt to invoke `compareTo` on `null`. If the references stored in the array are in ascending order, your method should return `null`.

27) The effect of the method written for Step 18 may be to scatter the value `null` in various indices in the array. Rearrange the references in the array so that all `null` references follow all non-`null` references.

28) Rewrite the method for Step 26 so that it calls the method for Step 27 as a first step and then stops checking for ascending order when a `null` reference is found in the array.

29) Run the complete program.

30) Copy the output in the Console view and paste it into a word processing document.

31) Print a hard copy of your output.

32) Print the file `GenericMethods.java`.

33) Print the file `TestGenericMethods.java`.

34) Submit your hard copies of the code and the output to the lab instructor.

35) Drag a copy of your `GenericMethods` folder from the **Desktop** onto a flash drive or a file server.

36) In the `Package Explorer` view select your `GenericMethods` folder.

37) Select **Edit > Delete**.

38) In the dialog box that opens, click the radio button that indicates you want to delete everything before you click the **Yes** button.

39) Quit Eclipse.

13

Files

INTRODUCTION

In all of the programs we have written so far, the user has interacted with the program by entering text or making choices by generating events through GUI components. We have not written any programs where the data from one execution of a program needs to be saved until the same program or another program uses that data at a later time. The memory unit (RAM) of a computer can store data while the program is executing, but not between executions. It is the storage devices of a computer—hard disk, CD, DVD, flash drive, tape, etc.—that are used to store data in a more permanent way that does not require continuous electrical power. Data stored on such media is usually organized in a hierarchical system. Users view folders as containers for files and other folders. On a UNIX system, *directories* take the place of folders as the means of organizing files. We will use the word "directory" below instead of "folder" to conform with the API.

Our goal in this chapter is to introduce the techniques for handling files in Java. The input/output (I/O) facilities in Java are provided by the API through importing `java.io.*`. There are many ways to store data in files and retrieve data from files in Java. We will not discuss all of the possibilities.

Aside from the actions of storing data in files and retrieving data from files, the Java API also provides the programmer with facilities for manipulating and inspecting the hierarchical organization of the storage system. We will first discuss these facilities and then move on to writing data into files and reading data from files.

The `File` class

`File` objects in Java are used to inspect and change file related information. `File` objects allow the programmer to access and modify attributes of files, but not the contents

(data) stored in the files. A *pathname* describes the name of a file and its position in the hierarchical organization of files. For example, if a hard disk is named `Frumplemeyer`, a folder at the top level of that hard disk is named `Examples`, and the folder named `Examples` contains a file named `Sample`, then the pathname of this file under Mac OS X is

```
Frumplemeyer/Examples/Sample
```

As with any object in Java, a `File` object must be constructed before it can be used. The four constructors provided by the API have the following signatures:

```
File (File parent, String child)
File (String pathname)
File (String parent, String child)
File (URI uri)
```

In the simplest case, the source code for the program is in the folder `Examples` along with the file `Sample`. In that case, the programmer supplies just the file name (which is the same as the pathname) as an argument to the `File` constructor. For example, to construct a `File` object for the file `Sample` described above, we could write the following code in your Java class:

```
File fileObj = new File("Sample");
```

We could then examine and/or manipulate the file's attributes using some of the methods of the `File` class described below.

`public boolean exists()`	Returns `true` if there is an item in the hierarchical storage system corresponding to the information specified when the `File` object was constructed.
`public boolean isFile()`	Returns `true` if the item is a file
`public boolean isDirectory()`	Returns `true` if the item is a directory
`public boolean canRead()`	Returns `true` if the item can be read by the user
`public boolean canWrite()`	Returns `true` if the item can be written by the user
`public long length()`	Returns the number of bytes in the item
`public long lastModified()`	Returns information regarding when the item was last modified
`public boolean delete()`	Returns `true` after successfully deleting the item from the storage system
`public boolean mkdir()`	Returns `true` after successfully creating a new directory in the storage system

The next sample program shows some simple uses of the methods listed above. For a little change, we use `BorderLayout`, `GridLayout`, and `JPanel` objects. Note that we have

made the class that **extends JFrame** also implement **ActionListener**. There is one **JButton** object, **testB**, who needs a registered **ActionListener**. It is simply a shortcut to use the **TryFile** object itself as the registered **ActionListener** for the **testB**. To accomplish this registration, we use the reserved word **this** to refer to the **TryFile** object being constructed and the class **TryFile** contains an **actionPerformed** method that describes the response to clicking **testB**. We have highlighted the lines of code that make this object its own handler below.

```
import java.io.*;  // to get the File class
import java.awt.*; // to get window components
import java.awt.event.*; // to get event handling
import javax.swing.*; // to get Swing GUI components

public class TryFile extends JFrame implements ActionListener  {

// Panel where label and text field for input will be placed
    private JPanel inputP;

    private JLabel promptL // label for input text field
    private JTextField inputTF; // text field for input

// Panel where output will be displayed in a text area with scroll pane
    private JPanel outputP;

    private JLabel resultsL; // to label output scroll pane
    private JScrollPane outputSP; // view of the output is controlled by this
                                  // scroll pane
    private JTextArea outputTA;// output will appear here
    private JButton testB; // will cause testing of the name entered
                           // in the input text field
    private Container myCP; //to hold a reference to the content pane

    public TryFile() {
        super("Try File Methods");
        myCP = getContentPane();
        // We use the default BorderLayout manager for the content pane
        setSize(400,300);
        setLocation(100,100);

        // Construct the panel for the input and add to the content pane
        inputP = new JPanel(new GridLayout(2,1));
        myCP.add(inputP,BorderLayout.NORTH);

        // Construct the GUI components that go on the input panel and add
        // them to the panel inputP.
        promptL = new JLabel("Enter the File name here:");
        inputP.add(promptL);
        inputTF = new JTextField(30);
```

```
        inputP.add(inputTF);

        // Construct the button and add it to the content pane
        testB = new JButton ("Test File");
        myCP.add(testB,BorderLayout.WEST);
        testB.addActionListener(this)   ;

        // Construct the panel for the output and add to the content pane
        outputP = new JPanel(new GridLayout(2,1));
        myCP.add(outputP,BorderLayout.CENTER);

        // Construct the GUI components that go on the output panel and add
        // them to the panel outputP.
        resultsL = new JLabel("Test Results", JLabel.CENTER);
        outputP.add(resultsL);
        outputTA = new JTextArea();
        outputSP = new JScrollPane(outputTA,
            JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
        outputSP.setSize(200,100);
        outputP.add(outputSP);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                   System.exit(0);
            }//windowClosing
        }); //end of definition of WindowAdapter and semicolon to end the line
        setVisible(true);
    }//constructor for TryFile

    public void actionPerformed(ActionEvent e) {
        String userInput = inputTF.getText();
            if(!userInput.equals("")){
                // Declare and construct File object
                File dataFile = new File (userInput);
                // Invoke various instance methods of the File class
                // We make use of the  ternary operator ( ? : ) to select
                // appropriate messages to display
                if(dataFile.exists()) {
                   outputTA.setText( dataFile.getName() + " exists\n" +
                        (dataFile.isFile() ? " is a file\n"
                             :" is not a file\n") +
                        (dataFile.canRead() ? " can be read\n"
                             :" cannot be read\n") +
                        (dataFile.canWrite() ? " can be written\n"
                             :" cannot be written\n"));
                }else { // item did not exist in the file system
```

```
                outputTA.setText (userInput + " does not exist");
            }//inner else
        } else {
            outputTA.setText ("You must enter a file name before clicking Test
            File.");
        }// outer else
        inputTF.setText("");
        inputTF.requestFocus();
    }//actionPerformed

    public static void main (String args[]) {
        TryFile myAppF = new TryFile();
    }//main method
}//class TryFile
```

Storing Data In Files and Retrieving Data from Files

Data can be stored in files in several formats. It is possible to store data in human readable characters. However, if this data is needed subsequently, it must be converted to its binary representation before another program can use it. Data can more efficiently be stored in its binary representation so that no conversion is needed in subsequent uses of the data by programs.

Data that is organized into objects during execution can be stored in a file that maintains that organization so that the objects can be reconstructed when needed. Code written to implement this strategy is actually fairly simple because Java takes care of converting each object into a stream of bytes that is written in a file and of later reversing the process to form a stream of bytes read from the file and from which the object is reconstructed. The formal name for the process of converting an object to a stream of bytes is *object serialization*. In order to make an object serializable, a programmer needs only to specify that the class of the object implements the interface `java.io.Serializable`. There are no methods to be implemented in the `Serializable` interface. We will continue to discuss object serialization in the next example.

Example—A Database of Phone Numbers

We will write an application that maintains a database of the home, cell, and work phone numbers of a set of people. Information can be added to or deleted from the database only while the program is running. Between executions of the program, the data will be stored in a file. The first time a user employs this program, the database will be empty. The user will enter data into the database through a GUI. When the user has completed a session of interaction with the database, she will have the option of saving all of the data in a file with a name she specifies. In a subsequent execution of the program, the user may choose to load data from a file into the database instead of beginning with an empty database.

We will begin by writing the class that represents the information about one person. We have chosen to name that class PhoneInfo.

The PhoneInfo Class

Each PhoneInfo object will have four private instance variables, one of class Name and three of class String. These variables will represent a person's:

1) name
2) home phone number
3) cell phone number
4) work phone number

We will use the Name class from Chapter 11 to represent each person's name. We will use a String object to represent each phone number. We will provide a five-parameter constructor to be used for entering data for the first and family names and all three phone numbers. We will also provide a two-parameter constructor to be used when a user is searching the database for phone information and can only supply the person's first and family names. We will also provide access and modifier methods for each instance variable and a toString method that returns a String representation of all of the information contained in a PhoneInfo object, a compareTo method so that class PhoneInfo will implement the Comparable<PhoneInfo> interface, and an equals method to override the equals method in the Object class. Remember that we specify that our class PhoneInfo implements java.io.Serializable so that we will be able to store information from these objects in a file and later reconstitute these objects. Below is the code for class PhoneInfo.

```
package forPhoneDB;

public class PhoneInfo implements Serializable, Comparable<PhoneInfo> {
        private Name myName; // represents the person's name
        private String myHomePhone; // represents the person's home phone number
        private String myCellPhone; // represents the person's cell phone number
        private String myWorkPhone; // represents the person's work phone number

            public PhoneInfo(String theFirst, String theFamily,
                String theHomePhone, String theCellPhone, String theWorkPhone) {
             myName = new Name(theFirst,theFamily);
             myHomePhone = theHomePhone;
             myCellPhone = theCellPhone;
             myWorkPhone = theWorkPhone;
    }// 5 parameter constructor

    public PhoneInfo(String theFirst, String theFamily) {
             myName = new Name(theFirst,theFamily);
             myHomePhone = "none";
             myCellPhone = "none";
             myWorkPhone = "none";
    }// 2 parameter constructor
```

```
    public Name getName() {
        return myName;
    }//getName

    public String getHomePhone() {
        return myHomePhone;
    }//getHomePhone

    public String getCellPhone() {
        return myCellPhone;
    }//getCellPhone

    public String getWorkPhone() {
        return myWorkPhone;
    }//getWorkPhone

    public void setName(Name theName) {
        myName = theName;
    }//setName

    public void setHomePhone(String theHomePhone) {
        myHomePhone = theHomePhone;
    }//setHomePhone

    public void setCellPhone(String theCellPhone) {
        myCellPhone = theCellPhone;
    }//setCellPhone

  public void setWorkPhone(String theWorkPhone) {
    myWorkPhone = theWorkPhone;
  }//setWorkPhone

    public int compareTo(PhoneInfo theOther){
            return myName.compareTo(theOther.myName);

    }//compareTo

    public boolean equals (Object compareObj) {
        PhoneInfo thePI = (PhoneInfo)compareObj;
        return (myName.equals(thePI.myName));
    }//equals

  public String toString() {
    return "Name: " + myName + "\n"
        + "Home Phone Number: " + myHomePhone + "\n"
        + "Cell Phone Number: " + myCellPhone + "\n"
        + "Work Phone Number: " + myWorkPhone + "\n";
  }//toString

}//class PhoneInfo
```

The GUI for the Database of Phone Numbers

The interface for our phone number database could have the following appearance:

Specifications for User Interaction with the Database

1) When the user wants to enter a new record in the database, she will type information in the first five text fields and press the **Enter** button. If information about the specified person already exists in the database, a message and the current information will be displayed. All buttons except the **OK** and **Cancel** buttons will be disabled until the user makes her choice. The user will press the **OK** button to indicate that she wants to replace the information currently in the database with the data she has just typed or she will press the **Cancel** button to indicate that the current information should remain unchanged in the database.

2) When the user wants to view all of the information in the database, she will press the **Display** button and the contents of the database will be displayed in the scroll pane at the bottom of the window.

3) When the user wants to search for a particular person's phone number, she will enter the person's family name and first name in the appropriate text fields and

press the **Search** button. If the person's information is found in the database, it will be displayed. If the person's information is not found in the database, an appropriate message will be displayed.

4) When the user wants to delete the information about a particular person from the database, she will enter the person's family name and first name in the appropriate text fields and press the **Delete** button. If the person's information is not found in the database, an appropriate message will be displayed. If the person's information is found in the database, the current information will be displayed and the user will be asked to confirm the delete request. All buttons except the **OK** and **Cancel** buttons will be disabled until the user makes her choice. The user will press the **OK** button to indicate that she wants to delete the information currently in the database or she will press the **Cancel** button to indicate that the current information should remain in the database.

5) When the user wants to load information from a file into the database, she will enter the filename in the appropriate text field and press the **Load** button. If there is any problem reading data from the file, an appropriate message will be displayed. If the file contains information about a person already in the database, the person's data from the file will *not* be added to the database and an appropriate message will be displayed. (This is an arbitrary design decision to simplify coding the program at this point.)

6) When the user wants to save information from the database into a file, she will enter a file name in the appropriate text field and press the **Save** button. If there is any problem saving the data in the designated file, an appropriate message will be displayed. In particular, if there already exists a file with the specified name, a message will be displayed asking the user if she wants to overwrite the file. All buttons except the **OK** and **Cancel** buttons will be disabled until the user makes her choice. She will press the **OK** button to overwrite the file or she will press the **Cancel** button to cancel the save request.

Instance Variables in the PhoneDB Class

Our `PhoneDB` class has the following instance variables that are references to GUI components:

```
private JLabel familyNameL, firstNameL, homePhoneL, cellPhoneL,
                                    workPhoneL, fileNameL, messageL;
private JTextField familyNameTF, firstNameTF, homePhoneTF, cellPhoneTF,
                                     workPhoneTF, fileNameTF;
private JScrollPane messagesSP;
private JTextArea messagesTA;
private JButton enterB, searchB, deleteB,
           displayB, loadB, saveB, okayB, cancelB;
```

Our PhoneDB class has several additional instance variables. Note that these variables are declared as instance variables because the inner handler classes need references to them.

```
private List<PhoneInfo> myPhoneList;//refers to the DB in memory
private String fileName;//refers to a file name being processed
                        // by a save or load command
private PhoneInfo currentPIRecord, found;// refer to a record currently being
                        //entered and a record found in the list in memory
private boolean processingSave,//set to true when awaiting OK or Cancel
                // during a Save to file operation
          duplicateRecord,//set to true when awaiting OK or Cancel
                // during a new data Enter operation
          processingDelete,//set to true when awaiting OK or Cancel
                // during a delete operation
private Container myCP;//handy reference to content pane
private String errorMsg, // to pass message from validNameInput method
          fName, //to pass first name from validNameInput method
          lName;//to pass family name from validNameInput method
```

Code in the Constructor for Class PhoneDB

There are many lines of code needed to get all of the GUI components constructed and functioning properly that we will not present here. The inner classes that act as handlers for the **Save** and **Load** requests are of primary interest in this chapter. We have, however, made the program a bit more interesting by employing the **OK** and **Cancel** buttons in three distinct ways. They are used to confirm or cancel **Save** requests when a file may be overwritten. They are used to confirm or cancel **Enter** requests when the user attempts to enter new information for a person who already exists in the database. They are also used to confirm or cancel **Delete** requests. We have supplied three boolean instance variables, processingSave, processingDelete, and duplicateRecord, to serve as flags indicating the situation we are handling with the **OK** and **Cancel** buttons. These instance variables need to be initialized in the constructor. The code below shows those initializations along with the construction of the PhoneInfoList object.

```
myPhoneList = new List< >(); // construct empty phone info list
processingSave = false; //not processing a save yet
duplicateRecord = false; //not processing a new entry yet
processingDelete = false; //not processing a delete yet
```

Some Utility Methods in the PhoneDB Class

We will find that at times the same lines of code need to be executed in more than one place in a program. When that happens, we should write the code once as the body of a method and then simply invoke that method wherever it is required. There are five

such methods in the PhoneDB class. The first method simply clears all of the data input text fields by setting the text to an empty String:

```
private void clearInputFields() {
    familyNameTF.setText("");
    firstNameTF.setText("");
    homePhoneTF.setText("");
    cellPhoneTF.setText("");
    workPhoneTF.setText("");
    fileNameTF.setText("");
}//clearInputFields
```

The second method is used to enable and disable sets of buttons. When the **OK** and **Cancel** buttons are enabled, all other buttons should be disabled. When the **OK** and **Cancel** buttons are disabled, all other buttons should be enabled. The method below has one boolean parameter and uses that value as the argument to setEnabled on all buttons except **OK** and **Cancel**. It uses the logical opposite of its parameter as the argument to setEnabled on the **OK** and **Cancel** buttons. Hence, if we need all buttons except **OK** and **Cancel** enabled, we invoke adjustButtons(true). In particular, we do this in the constructor as a form of initialization of the status of the buttons. If we need all buttons except **OK** and **Cancel** disabled, we invoke adjustButtons(false).

```
private void adjustButtons(boolean tFValue) {
    saveB.setEnabled(tFValue);
    enterB.setEnabled(tFValue);
    displayB.setEnabled(tFValue);
    searchB.setEnabled(tFValue);
    deleteB.setEnabled(tFValue);
    loadB.setEnabled(tFValue);
    okayB.setEnabled(!tFValue);
    cancelB.setEnabled(!tFValue);
}//adjustButtons
```

The third method is used to invoke both clearInputFields and adjustButtons(true). We have named it reset to indicate that the GUI is being reset to its original status.

```
private void reset() {
    adjustButtons(true);
    clearInputFields();
}//reset
```

The fourth method is used to process the user's input of a first or a last name. getUserInput returns the String that is returned by invoking getText on the JTextField that is its first parameter. If the user has not entered any text, the method uses its second String parameter to compose an error message that is concatenated to the reference named errorMsg. That potential change to errorMsg is technically a *side effect* since the variable may be effected by the action of the method, but that effect cannot

be decerned from the signature or return type of the method. Our method getUserInput is displayed below.

```
private String getUserInput(JTextField theTF, String theText){
    String userInput = theTF.getText();
    if (userInput.equals("")){
            errorMsg += "You need to enter a " + theText + " name.\n";
    }//if
    return userInput;
}//getUserInput
```

Our fifth method employs the method getUserInput twice to process the input for the first name and then the input for the family name. It returns a boolean value of true when both text fields have non-empty Strings. It also produces two side effects. It assigns the first name and family name returned by getUserInput to instance variables fName and lName.

```
private boolean validNameInput() {
    errorMsg = "";
    fName = getUserInput(firstNameTF, "first");
    lName = getUserInput(familyNameTF, "family");
    return errorMsg.equals("");
}//validNameInput method
```

Inner Classes For Event Handling

Class EnterBHandler

Let us look at the EnterBHandler class first to see how the boolean flag duplicateRecord is set when it is needed. Consider the code below.

```
public class EnterBHandler implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    if (validNameInput()){
      currentPIRecord = new PhoneInfo(fName, lName,
        homePhoneTF.getText(), cellPhoneTF.getText(), workPhoneTF.getText());
      found = myPhoneList.search(currentPIRecord);
      if (found != null) {//person already in DB
        duplicateRecord = true;
        adjustButtons(false);
        messagesTA.setText("\nRecord already exists:\n"+ found.toString()
              + "\nPress OK to replace old record " +
                   "or Cancel to cancel new entry.\n");
      } else {
        if(myPhoneList.add(currentPIRecord)) {
          messagesTA.setText("\n" + currentPIRecord + "added to the DB.\n");
        }else {
```

```
            messagesTA.setText("\nFailed to add " +
                    currentPIRecord + " to the DB.\n");
          }//else
        }//else for add new record
        clearInputFields();
      } else {
        messagesTA.setText(errorMsg + "\n");
      }//else
    }//actionPerformed
}//EnterBHandler
```

Note that the code above constructs a `PhoneInfo` object from the data entered by the user and then searches the current database to see if the specified person already has a record in the database. If the reference `found` is not `null`, we need to inform the user and wait for another event, pressing `OK` or `Cancel`, before we change the data in the database. The three actions necessary in the code are:

1) Assign `true` to the flag `duplicateRecord`.
2) Disable all buttons except **OK** and **Cancel**
3) Display a message informing the user about the issue at hand.

The lines of code highlighted above accomplish these actions.

Class SaveBHandler

When the user issues a **Save** request, it is always good practice to check that the **Save** operation will not inadvertently erase a useful file by overwriting it. Hence, the `actionPerformed` method of class `SaveBHandler` should construct a `File` object and inform the user of any problems or potential overwriting. If the user supplies a file name that does not correspond to an existing entity, the code can simply perform the **Save** operation by commanding `myPhoneList` to save itself to a file with the specified name. It is also useful to have the method that performs the save process return a `String` object that contains text describing any error conditions encountered during the save process. Hence, code similar to the following might be used.

```
String message = myPhoneList.saveToFile(fileName);
messagesTA.append("Data saved to file "+fileName+".\n"
                                  + message + "\n");
```

If the name for the file happens to be the name of an already existing directory instead of a file, the user should be informed. If the file exists and cannot be overwritten, the user should be informed. Finally, if the file exists and can be overwritten, the user should be given a chance to bail out of the situation. The program supplies this last chance to the user by forcing the user to press **OK** or **Cancel** before any actions to overwrite the file take place. The buttons other than **OK** and **Cancel** need to be disabled, the `boolean` flag `processingSave` needs to be assigned the value `true` so that the **OK** and **Cancel** button handlers know the situation, and finally the user needs to be informed of the situation. The following three lines of code accomplish the tasks just described.

```
    adjustButtons(false);
    processingSave = true;
    messagesTA.appendText("Press OK to overwrite file "+
        fileName + " or press Cancel to cancel save request\n");
```

Consider the code of class `SaveBHandler` displayed below with the lines involved in the file processing highlighted.

```
public class SaveBHandler implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    fileName = fileNameTF.getText();
    fileNameTF.setText("");
    String message = "";
    if(fileName.compareTo("") > 0){
      File theFile = new File(fileName);
      if(!theFile.exists()) {
        message = myPhoneList.saveToFile(fileName);
        messagesTA.setText("Data saved to file "+fileName+".\n"
            + message + "\n");
      } else if (theFile.isDirectory()){
        messagesTA.setText("Error: " + fileName + " is a directory.\n");
      } else if (!theFile.canWrite()) {
        messagesTA.setText("Cannot write data to "+fileName+".\n");
      } else {
        adjustButtons(false);
        processingSave = true;
        messagesTA.setText("\nPress OK to overwrite file "+
          fileName + " or press Cancel to cancel save request\n");
      }//else
    } else {
      messagesTA.setText("You must enter a file name in order to save a file");
    }//else
  }//actionPerformed
}//SaveBHandler
```

Note that we have assumed that a `PhoneInfoList` object knows how to save itself to a file when provided with the file name as an argument to its `saveTofile` method as in the line of code

```
message = myPhoneList.saveToFile(fileName);
```

Both the `SaveBHandler` and `EnterBHandler` methods may pass control of the situation to the `actionPerformed` methods of the `OKBHandler` and `CancelBHandler` objects. We will address this issue after we finish the **Save** operation.

The saveToFile Method

We need to see just how the **Save** operation is coded. We have assumed that `myPhoneList` is of class `List<PhoneInfo>`. Hence, the `saveToFile` method is needed in the `List<E>` class.

We need to connect two streams of bytes to accomplish moving the data from memory to a file. We will construct a `FileOutputStream` object using the file name as an argument. The `FileOutputStream` object is a conduit through which bytes flow to a file. We must construct an `ObjectOutputStream` through which the bytes of an object flow from memory to the `FileOutputStream`. In addition, we want to preserve the structure of our data as `PhoneInfo` objects. In order to do that, we needed to declare that the class `PhoneInfo implements java.io.Serializable` and that class `Name implements java.io.Serializable.` The following diagram describes the data conduit we will construct.

`phoneInfo object in memory→ObjectOutputStream→FileOutputStream→file`

There are five constructors for `FileOutputStream` objects provided in the API. Their signatures are as follows:

`FileOutputStream(File)`

Creates a file output stream to write to the file represented by the specified `File` object.

`FileOutputStream(File, boolean)`

Creates a file output stream to write to the file represented by the specified `File` object.

`FileOutputStream(FileDescriptor)`

Creates an output file stream to write to the specified `FileDescriptor`, which represents an existing connection to an actual file in the file system.

`FileOutputStream(String)`

Creates an output file stream to write to the file with the specified name.

`FileOutputStream(String, boolean)`

Creates an output file stream to write to the file with the specified name.

We have chosen to use the fourth constructor in our code, simply passing the file name to the `FileOutputStream` constructor.

There are two constructors for an `ObjectOutputStream` object provided by the API. The signature of the one we will use is as follows:

`ObjectOutputStream(OutputStream)`

Creates an `ObjectOutputStream` that writes to the specified `OutputStream`.

In our code, we first construct the `FileOutputStream` object and pass it as an argument to the `ObjectOutputStream` constructor. We use an anonymous `FileOutputStream` and immediately pass it to the `ObjectOutputStream` constructor as follows:

```
ObjectOutputStream oOS = new ObjectOutputStream(
                                        new  FileOutputStream(fileName));
```

Some of the methods available in class `ObjectOutputStream` are listed below. Our code uses only `writeObject`, `flush`, and `close`.

close()—Closes the stream.
defaultWriteObject()—Write the non-static and non-transient fields of the current class to this stream.
flush()—Flushes the stream.
reset()—Reset will disregard the state of any objects already written to the stream.
write(byte[])—Writes an array of bytes.
write(byte[], int, int)—Writes a sub array of bytes.
write(int)—Writes a byte.
writeBoolean(boolean)—Writes a boolean.
writeByte(int)—Writes an 8 bit byte.
writeBytes(String)—Writes a String as a sequence of bytes.
writeChar(int)—Writes a 16 bit char.
writeChars(String)—Writes a String as a sequence of chars.
writeDouble(double)—Writes a 64 bit double.
writeFloat(float)—Writes a 32 bit float.
writeInt(int)—Writes a 32 bit int.
writeLong(long)—Writes a 64 bit long.
writeObject(Object)—Write the specified object to the ObjectOutputStream.
writeShort(int)—Writes a 16 bit short.
writeUTF(String)—Writes a String in UTF format.

Since various exceptions can be thrown in the process of constructing and using an ObjectOutputStream, we have simply chosen to take Peter Van Der Linden's advice and catch any Exception. The code for our method saveToFile is shown below. It uses the standard template for processing each entry in an array with a for loop. In the body of the loop, we write each object to the ObjectOutputStream oOS. It is good practice to flush any bytes out of the stream and close the stream when we have completed transmitting all of the bytes through writeObject.

```
public String saveToFile(String fileName) {
    String messageFromSave = "";
    try{
        ObjectOutputStream oOS = new ObjectOutputStream(
                new FileOutputStream(fileName);
        for(int i = 0; i < myCount  ; i++) {
            oOS.writeObject(myArray[i]);
        }//for
        oOS.flush();
        oOS.close();
    }//try
    catch(Exception e) {
        messageFromSave = e.toString();
    }//catch
    return messageFromSave;
}//saveToFile
```

Classes OkayBHandler and CancelBHandler

The structure of the code in the actionPerformed methods of classes OkayBHandler and CancelBHandler are similar in that they need to select between alternatives depending on which boolean flag is set. It is also the case that the **OK** and **Cancel** buttons should not generate events unless one of these flags is set to true. Hence, we have written the code below to test the flags to determine the appropriate actions and to send an error message to the Java Console view if somehow the code is handling an event for one of these buttons when it should be disabled, i.e., none of the flags is set to true. Consider first the code in the actionPerformed method of class CancelBHandler since it is simpler to cancel a request than to perform a request. Note that the appropriate boolean flag needs to be reset to false just after we acknowledge to the user that the **Save**, **Enter**, or **Delete** request has been cancelled. In addition, note that the enabled status of all of the buttons must be reversed and all text fields cleared as the last action this code performs by invoking the reset method.

```
public class CancelBHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if(processingSave) {
            messagesTA.setText("Save request cancelled. "
                       + fileName + "unchanged.\n");
            processingSave = false;
        } else if (duplicateRecord) {
            messagesTA.setText("Information about "+ fName
                       + " " +lName + " unchanged.\n");

            duplicateRecord = false;
        }else if(processingDelete){
            messagesTA.setText("Delete request cancelled. \n");
            processingDelete = false;
        } else {
            System.out.println("Cancel Button being handled at inappropriate time"
                + e.toString());
        }//else
        reset();
    }//actionPerformed
}//CancelBHandler
```

The code in the actionPerformed method of the OkayBHandler class is just a bit longer because it must contain the code to command the saveToFile or delete a record or add a record's replacement. Notice, however that the structure of decision making is the same. The boolean flags are used to select the appropriate code segment to execute.

```
public class OkayBHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if(processingSave) {
            String errmsg = myPhoneList.saveToFile(fileName);
            messagesTA.setText(fileName + " over written.\n"
                       + errmsg + "\n");
            processingSave = false;
        } else if (duplicateRecord) {
            if(myPhoneList.delete(found)) {
                if(myPhoneList.add(currentPIRecord)){
                  messagesTA.setText("\nRecord for " + currentPIRecord.getName()
                                     + " changed.\n");
                }else {
                  messagesTA.setText("\nError in adding new record. " +
                       fName + " " + lName + " deleted from DB.\n");
                }//else
            } else {
                messagesTA.setText("\nError in deleting old record. "
                                     "No change in DB.\n");
            }//inner else
            duplicateRecord = false;
        }else if (processingDelete) {
            if (myPhoneList.delete(found)) {
                messagesTA.setText("The record for " + found + " was deleted.");
            } else {
                messagesTA.setText("Failure occurred in deleting " + found + ".\n");
            }
            processingDelete = false;
        }else {
            System.out.println("OK Button being handled at inappropriate time"
                + e.toString());
        }//else
        reset();
    }//actionPerformed
}//OkayBHandler
```

Class LoadBHandler

Finally, we come to the class `LoadBHandler`. The `actionPerformed` method of class `LoadBhandler` should construct a `File` object using the file name supplied by the user and make sure that the file exists, that it is a file, and that it can be read before attempting to read the objects from the file and insert them into `myPhoneList`. The `actionPerformed` method should display an appropriate error message if the file either does not exist or cannot be read. In the case that the file exists and can be read, we need to construct the following layering of streams to accomplish our goal of reading and reconstituting the `PhoneInfo` objects stored in the file.

phoneInfo object in file → FileInputStream → ObjectInputStream → memory

A FileInputStream must be constructed before the ObjectInputStream can be constructed.

We need some kind of looping structure, like a while loop, to repeatedly read an object from the ObjectInputStream, cast that object to class PhoneInfo to regain its former structure, and insert that PhoneInfo object into myPhoneList.

There are three constructors in class FileInputStream. Their signatures are listed below. In our code, we have chosen simply to pass the String that refers to the file name to the FileInputStream constructor.

FileInputStream(File)

Creates a FileInputStream by opening a connection to an actual file, the file named by the File object in the file system.

FileInputStream(FileDescriptor)

Creates a FileInputStream by using the file descriptor, which represents an existing connection to an actual file in the file system.

FileInputStream(String)

Creates a FileInputStream by opening a connection to an actual file, the file named by the path in the file system.

There are two constructors in class ObjectInputStream. The signature of the constructor we use is as follows:

ObjectInputStream(InputStream)

Creates an ObjectInputStream that reads from the specified InputStream.

Below are some of the methods available in class ObjectInputStream. Our code uses only the readObject method.

available()—Returns the number of bytes that can be read without blocking.
close()—Closes the input stream.
defaultReadObject()—Read the non-static and non-transient fields of the current class from this stream.
read()—Reads a byte of data.
read(byte[], int, int)—Reads into an array of bytes.
readBoolean()—Reads in a boolean.
readByte()—Reads an 8 bit byte.
readChar()—Reads a 16 bit char.
readDouble()—Reads a 64 bit double.
readFloat()—Reads a 32 bit float.
readFully(byte[])—Reads bytes, blocking until all bytes are read.
readFully(byte[], int, int)—Reads bytes, blocking until all bytes are read.
readInt()—Reads a 32 bit int.

`readLine()`—Reads in a line that has been terminated by a `\n, \r, \r\n` or `EOF.`
`readLong()`—Reads a 64 bit `long.`
`readObject()`—Read an object from the `ObjectInputStream.`
`readShort()`—Reads a 16 bit `short.`
`readUnsignedByte()`—Reads an unsigned 8 bit byte.
`readUnsignedShort()`—Reads an unsigned 16 bit `short.`
`readUTF()`—Reads a UTF format `String.`
`registerValidation(ObjectInputValidation, int)`—Register an object to be validated before the graph is returned.
`skipBytes(int)`—Skips bytes, block until all bytes are skipped.

We should note another technique used in our coding of the method `loadFromFile`. We have written what appears to be an infinite loop headed with `while(true)`. However, we know that the file will have finite length. After the last object has been read, the invocation of `readObject` will fail and throw an `EOFException` (end of file). The code of `loadFromFile` is ready to catch that `EOFException`. In fact, our code depends on that `EOFException` being thrown and caught to make the program work, and we never write an instruction to close the input stream.

We want to display an error message for the user regarding any unusual condition encountered as the file is read object by object. Hence, the return type of method `loadFromFile` will be `String`. The method will begin with a local variable named `toReturn` initialized to an empty `String`. If any record from the file fails to be added to the database, the information about that record will be appended to the `String toReturn`. If an `Exception` is thrown as the file is read, and that `Exception` is not an `EOFException`, the `String` representation of the `Exception` will be appended to the `String toReturn`. The last line of code of `loadFromFile` returns a reference to `toReturn`. Hence, the `actionPerformed` method of class `LoadBHandler` can test the value of the returned `String` to determine whether any unexpected behavior occurred or if an `Exception` resulting from an error condition was thrown.

The method `loadFromFile` has one parameter that is the `String` reference to the file name entered by the user. Our method `loadFromFile` in declared in the class `List<E>` and is displayed below.

```
public String loadFromFile(String fileName) {
  String toReturn = "";
  try {
    ObjectInputStream oIS =
      new ObjectInputStream(new FileInputStream(fileName));
    while(true) {
      E fromFile = (E)(oIS.readObject());
      E found = search(fromFile);
      if( found == null) { //object not already in List
        if (add(fromFile)) {
          toReturn += fromFile + " successfully added to List.\n";
        }else {
```

```
            toReturn += fromFile + " not successfully added to List.\n";
          }//inner else
        }else {
          toReturn += found + " already in DB.\n"
                   + "record not added from file!\n";
        }//else
      }//while
    }//try
    catch(EOFException eOF) {
    }//catch
    catch(Exception e) {
      toReturn += e;
    }//catch
    return toReturn;
}//loadFromFile
```

Finally, we will display the code of the `LoadBHandler` class below. Note how we invoke the method `loadFromFile` on `myPhoneList` and how we test the returned `String`.

```
public class LoadBHandler implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    fileName = fileNameTF.getText();
    if(fileName.compareTo("") > 0){
      File theFile = new File(fileName);
      if(!theFile.exists()) {
           messagesTA.setText(fileName +
                  " does not exist—cannot load data\n");
      }else if(!theFile.canRead()) {
        messagesTA.setText("Cannot read from " + fileName+"\n");
      } else {
        String fromLoad = myPhoneList.loadFromFile(fileName);
        messagesTA.setText("Data loaded from "+fileName+"\n"
                  + fromLoad + "\n");
      }//else
      clearInputFields();
    } else {
      messagesTA.setText("You must enter a file name " +
                    "in order to load a file");
    }//else
  }//actionPerformed
}//LoadBHandler
```

Exercises 13.1 and 13.2 : Extensions of Exercises 12.4 and 12.5

For Exercise 13.1, use the database of address information application described in Exercise 11.10 and continued in Exercise 12.4. For Exercise 13.2, use the database of birthday information described in Exercise 11.12 and continued in Exercise 12.5.

Modify both the front end and the back end of the database application so that:

1) The user may save the data to a file she specifies. In addition, the program should:
   a) Warn the user if she is about to overwrite an existing file, and give the user the option to cancel the save request if she does not wish to overwrite the existing file.
   b) Warn the user if the file is exists and is locked, and hence cannot be overwritten.
   c) Provide the user with an appropriate message if saving to the file is successful.
2) The user may read load data from a file. The program should provide the user with an appropriate message in each of the following situations:
   a) The file does not exist.
   b) The name specified is a directory name and not a file name.
   c) Reading from an existing file is not possible.
   d) Reading from the requested file is successful.

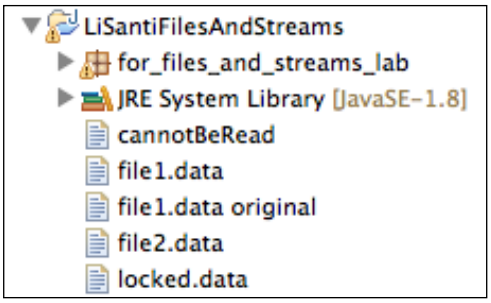
Exercises 13.3 and 13.4: Extensions of Exercises 13.1 and 13.2

Instead of writing each object in the `List<E>` to a file one at a time and later adding them back from the file into another `List<E>` one at a time, serialize the entire `List<E>`. Then simply write the entire list to the file with one line of code when the data is to be saved and read the entire list from the file when the data is to be loaded.

Lab: Files and I/O Streams • Chapter 13

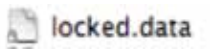
The objective of this lab is to learn about some of the methods of the `File` class supplied as part of the Java API in the package `java.io`. These methods provide a way to give reasonable error messages to the user when difficulty is encountered during file processing.

1) Get a copy of the folder named `for_files_and_streams_lab` onto the **Desktop**.
2) Get a copy of the folder `dataForFilesAndStreams` onto the **Desktop**.
3) You will run the application that is coded into `TryFileClass`. The purpose of the application is to demonstrate some of the information available to you through calls to methods of the `File` class provided as part of the `java.io` package. The methods used in this lab are `exists()`, `isFile()`, `canRead()`, `getName()`,and `canWrite()`. The purposes of these methods are somewhat obvious from their names, but below you will be asked to write a sentence describing each method's usefulness.
4) Launch Eclipse, create a Java project entitled with `your name` and `FilesAndStreams`, and add the folder `for_files_and_streams_lab` to the project. In the pictures below and in the text of this handout, my Eclipse project folder is named `LiSantiFilesAndStreams`.
5) Select all files WITHIN the folder `dataForFilesAndStreams` and add them to the project. You should have a project structure similar to the one shown below.



6) In the **Finder**, double click the folder that Eclipse created for your project to open its window.
7) Select the file named `locked.data`.
8) Select **Get Info** from the **File** menu.
9) Check the locked check box. We should find later that this file cannot be overwritten.

10) A small padlock icon should appear on the file's icon in the **Finder** window.



11) Select the file named `file2.data`.

12) Select **Get Info** from the **File** menu.

13) Check the locked check box. We should find later that this file cannot be overwritten. A small padlock icon should appear on the file's icon in the **Finder** window.

14) Launch the Terminal application.

15) Change directories into your Eclipse project folder by typing the following at the prompt (with your Eclipse project folder name substituted for mine) :

```
cd Desktop/LiSantiFilesAndStreams
```

16) List the contents of the folder with a display of the permissions on each item by typing the following at the prompt:

```
ls -la
```

17) `rwxrwxrwx` means that the user (you), the group, and any other user have read, write, and execute permissions. Change your permission on the file named `cannotBeRead` by typing the following at the prompt:

```
chmod u-r cannotBeRead
```

`u-r` stands for "user minus read". You are doing this to create a file in your folder that you cannot read.

18) Type the `ls -la` command again and note that the first `r` in the permissions on file `cannotBeRead` has been replaced with a dash.

19) Type `logout` in the Terminal application's window.

20) Quit the Terminal application.

21) Within Eclipse's `Package Explorer` view, open the file `TryFileClass.java`.

22) Run the application.

23) Type the file name `for_files_and_streams_lab/TryFileClass.java` in the text field below the prompt.

24) Click the `Test` button.

> Q1) What information about the file `for_files_and_streams_lab/TryFileClass.java` is displayed in the text area?

25) Type the folder name `for_files_and_streams_lab` in the text field below the prompt.

26) Click the `Test` button.

> Q2) What information about the folder `for_files_and_streams_lab` is displayed in the text area?

27) Type the file name `locked.data` in the text field below the prompt.

28) Click the `Test` button.

> Q3) What information about the file `locked.data` is displayed in the text area?

29) Finally, you will try with a file name for a file that does not exist. Type your name in the text field below the prompt.

30) Click the `Test` button.

> Q4) What information about the nonexistent file having your name is displayed in the text area?

31) Quit the application by clicking the red button in the left end of the window's title bar.

32) Below you will answer some additional questions about the code in the file `TryFileClass.java`. Note that the class `TryFileClass` not only `extends JFrame`, but also `implements ActionListener`. Hence, the object constructed is its own handler. Read the code in the `actionPerformed` method of class `TryFileClass`, answer the following questions in complete sentences, and submit your work to the lab instructor.

> Q5) What is the class of objects that can invoke the methods `exists()`, `isFile()`, `canRead()`, `getName()`, and `canWrite()`?
>
> Q6) What is the `return` type of each of the methods `exists()`, `isFile()`, `canRead()`,`getName()`, and `canWrite()`?
>
> Q7) For each of the methods `exists()`, `isFile()`, `canRead()`, `getName()`, and `canWrite()` describe how to interpret the value it returns.

33) Close the editor view for TryFileClass.java.

34) Double click on the file TryStreams.java in the Package Explorer view to open the file and read the code of class TryStreams.

You will now run this application in several different situations. Some of those situations will throw exceptions that will be caught by the code already provided. Your eventual objective will be to provide more code in this program, using the File class, that will give appropriate messages to the user and avoid exception throwing.

35) Run the application.

36) Enter the file name file1.data in the text field below the prompt.

37) Click the Read File button in the SOUTH segment of the layout and note the message displayed.

38) Enter the file name file1.data in the text field below the prompt.

39) Click the Save File button in the SOUTH segment of the Layout and note the message displayed. The values in file1.data have been over written. The old values are lost forever!

40) Enter the file name file1.data in the text field below the prompt.

41) Click the Read File button . You get a message displaying the values that were read from the file and confirming that the original values are gone.

42) Enter the file name file2.data in the text field below the prompt.

43) Click the Read File button. You should see the message displaying the values that were read from the file.

44) Enter the file name file2.data in the text field below the prompt.

45) Click the Save File button. Note the message displayed because the file named file2.data is locked. Hence, you cannot over write its contents.

46) Enter the file name file2.data in the text field below the prompt.

47) Click the Read File button. Again, you should see the message displaying that values were read from the file. Note that these values have not changed since the previous read, indicating that as noted, the earlier request to save failed.

48) Enter the file name file3.data in the text field below the prompt.

49) Click the Read File button. Note the error message generated by the fact that file3.data does not exist!

50) Enter the file name file3.data in the text field below the prompt.

51) Click the Save File button. file3.data now exists.

52) Enter the file name file3.data in the text field below the prompt.

53) Click the Read File button. You should see the message displaying the values that were read from the file. This is additional confirmation that the last save was successful. Note that file3.data will appear in the Finder's window for your Eclipse project, but not in the Package Explorer view.

54) Quit the application by clicking the red button in the left end of the window's title bar.

55) Now you will add code in the `actionPerformed` method of the `ReadBHandler` class to provide the following more user friendly messages. In the `ReadBHandler` class, add code BEFORE the `try` and `catch` clauses to

   a) display an appropriate message and avoid throwing an exception and execution of the `catch` if the item associated with the filename does not exist.

   b) display an appropriate message and avoid throwing an exception and execution of the catch if an item associated with the filename exists, but is not a file.

   c) display an appropriate message and avoid throwing an exception and execution of the `catch` if the item exists and is a file, but cannot be read.

56) Run and debug the project. Make sure to test your program with:

   `file1.data`—No error messages should be generated in reading.

   `file2.data`—No error messages should be generated in reading.

   `for_files_and_streams_lab`—An error message should be generated when reading, indicating that the item exists, but is not a file.

   `frog.data`—An error message should be generated when reading, indicating the file does not exist.

Now you will add code in the `actionPerformed` method of the `SaveBHandler` class to provide the following more user friendly messages.

57) In the `SaveBHandler` class, add code BEFORE the `try` and `catch` clauses to

   a) tell the user that the file is being overwritten if the item already exists, and is a file.

   b) display an appropriate message and avoid throwing an exception and execution of the `catch` if the item exists and is a file, but cannot be over written.

   c) display an appropriate message and avoid throwing an exception and execution of the `catch` if the item exists and is not a file.

58) Run and debug the project. Make sure to test your program with:

   `file1.data`—A warning message should be generated in writing, indicating that the existing file is being overwritten.

   `file2.data`—An error message should be generated when writing indicating that the existing file cannot be overwritten.

   `for_files_and_streams_lab`—An error message should be generated when writing, indicating that this is a directory.

   `moose.data` —No error messages should be generated in writing (since the file does not exist).

59) Print a hard copy of the program, making sure that you have included your name in the comments, and submit it to the lab instructor.

60) Quit Eclipse.

61) Copy you're the edited `TryStreams.java` onto a flash drive.

14

Introduction to Linked Lists

INTRODUCTION

Why Not Just Use Arrays?

In the last few chapters we have explored the array construct that Java provides. Arrays provide a convenient way to represent the commonly used data structure called a *list*. Lists provide us with a means of maintaining several pieces of data, all of the same class or type. Many computer applications, including databases and word processors, rely heavily on the use of such lists. However, arrays are not the only data structure used to maintain a list. A second common representation for a list is called a *linked list*. Both types of lists have their advantages and disadvantages. However, it is useful now to look at some of the disadvantages of lists whose underlying representation uses an array.

When the JRE constructs an array, Java dynamically allocates a certain amount of memory for the array based on the argument we send to the array constructor. For example, the code below declares an array of `int`s named `myNumbers` and allocates 100 contiguous memory locations for `int` values.

```
int [ ] myNumbers = new int [100];
```

As another example, the code below declares an array of `String`s named `myStrings` and allocates 100 contiguous memory locations for references to `String` objects.

```
String[] myStrings = new String[100];
```

Once we have asked Java to allocate the space for the array, we cannot extend or shrink the size of the array. Our only option is to request a new allocation of space, and discard the currently allocated space. When is this an issue? If we are writing a program, we must anticipate the maximum number of positions in the array that we will need. If,

during execution of our program, we find that we need a larger array, we cannot extend the existing array. This is a problem, because presumably, the existing array is full, and we do not want to lose the data that is already maintains. To fix this problem, we must employ an algorithm something like the following:

When we need to expand the size of the array, because it has reached capacity:

- Declare a reference to a second array named `temp`.
- Construct an array that has more space allocated to it than the original array and make `temp` point to the second array.
- Copy the values stored in the original array into the second array.
- Make the reference to the original array point to the second array.

Although these steps seem simple, the overhead for copying the contents is rather high.

Exercise 14.1: Write a program that constructs a five element array of `String` references. The program should read `String`s from the user and store the references in the array until she indicates she has no more data to read. If the array becomes full, the program should have a method that doubles the capacity for `String`s that can be stored using the algorithm described above.

Moreover, because of the fixed size of an array, we must always be checking to determine if an action attempts to use an index that is out of range or otherwise invalid. For example, before we add an element to the array, we must make sure there is space for it. When we delete an element from an array, we have freed up space that should allow another element to be added.

Although accessibility of array elements is flexible—we can easily move from the 2nd element to the tenth element—manipulating the actual contents of the array requires a bit of work. Consider the issue of deleting an element from an array of 100 elements. Let us assume we wish to delete the element in position 50 and that the array currently holds 89 elements. We have two options. The first option is to clear out the value at position 50. Although this seems like an easy solution, it makes tasks such as adding to the array, sorting the array, and printing the array more difficult. For example, if we wish to add to the array, we must search for an empty spot in which to add. We no longer just add at the end of the array. Another solution is to move up elements in positions 51 through 88 to the next lower index position. This requires a lot of assignments, but makes methods that do such tasks as adding and sorting much easier. Neither solution is very elegant.

Linked lists have as some of their advantages flexibility for modification and for changes in size.

An Overview: Arrays vs. Linked Lists

Recall the class `List<E>` from Chapter 12, and consider the picture below of the object `myStringList` shown just after the list has been constructed with the following line of code:

```
List <String> myStringList = new List < > (100);
```

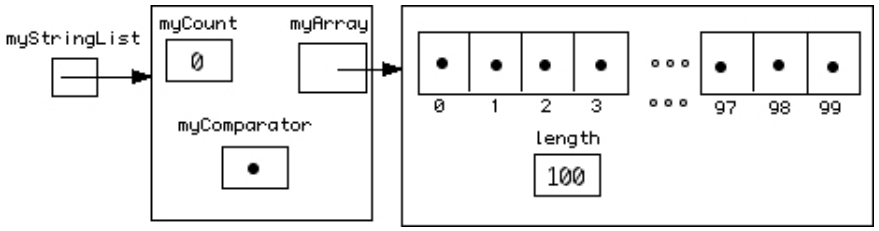


Figure 14.1 An empty list with an underlying array

In comparison, assume that we have written the Java code necessary to define a class `WCSLinkedList <E>`, and consider the following object, shown in Figure 14.2 below, constructed as a linked list of `String`s with the following line:

```
WCSLinkedList<String> myStringLinkedList = new WCSLinkedList< >();
```

Note that we have used the diamond operator of Java 1.7 in the line of code above. The compiler can infer the type parameter for the construction of the `WCSinkedList` from the type parameter used in the declaration of `myStringLinkedList`. Hence, we do not need to repeat the type parameter `String` within the angle brackets on the right side of the assignment operator.

This list has no counter, and, as yet, uses very little space in memory. It appears that the linked list object has one instance variable named `myHead` and its value is currently `null`. Exercise 14.21 adds an instance variable for a `Comparator` object to the `WCSLinkedList` class.

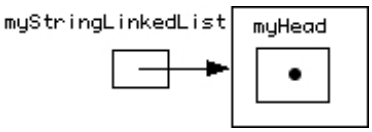


Figure 14.2 An empty list with an underlying linked list

Assuming that we have added five elements to our list that is implemented with an array, it would look as follows:

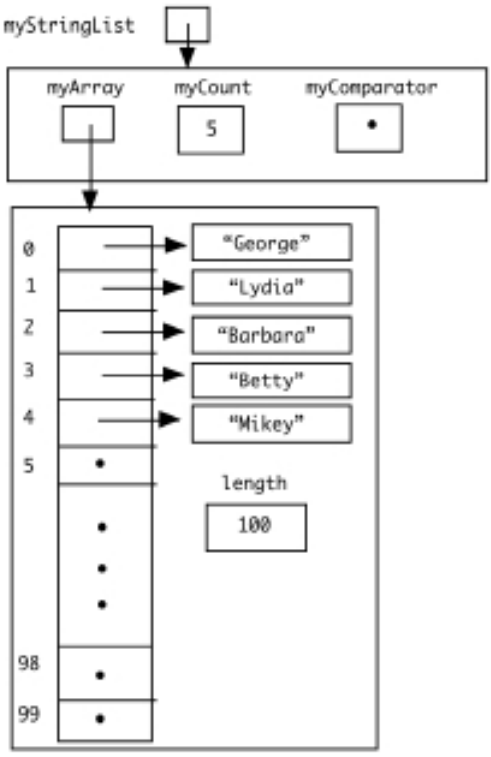


Figure 14.3 A list that contains 5 elements with an underlying array structure

As we add elements to our linked list structure, it will grow dynamically as space is needed. Hence, after adding the first element, it will look as in Figure 14.4 below:

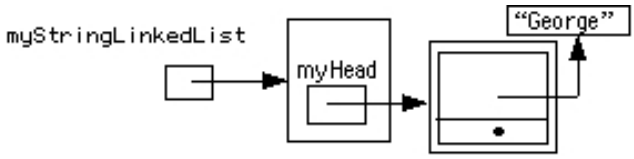


Figure 14.4 A list that contains 1 element with an underlying linked structure

After adding the second element to the linked structure, it appears as in Figure 14.5 below:

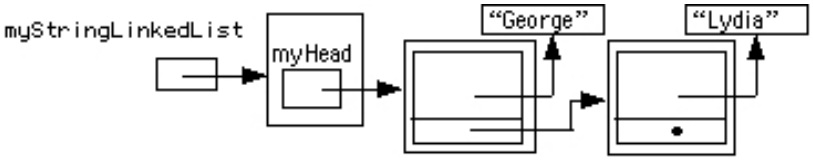


Figure 14.5 A list that contains 2 elements with an underlying linked structure

By the time we have added five elements, the linked structure will appear as in Figure 14.6 below:

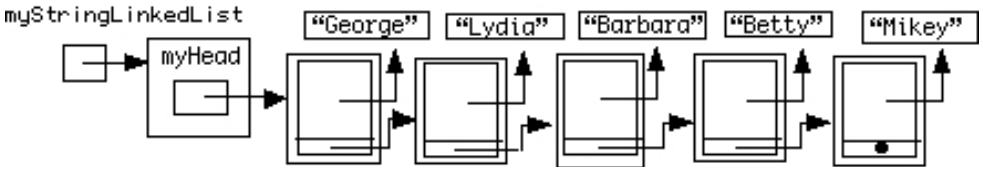


Figure 14.6 A list with an underlying linked structure that contains 5 elements

A linked list is made up of individual objects called *nodes*. Each node has two essential parts or *fields*: its data field and its link field. The data field of a node can be a pointer to an object (as shown above), or it can be a primitive value, like an `int`. The link field is a reference or *pointer* to another node. It is *important* to realize that the link field points to the entire node that follows. For example, the link field of the node containing a reference to `"George"` points to the entire node that contains a reference to `"Lydia".` The link out of the node containing `"George"` is *not* just pointing to the data field of the next node.

Notice that in our Figure 14.6, the link field in the last node in the linked list has a round dot, indicating that it is not pointing to anything. This reference is said to have the value `null`. If another item is added to the end of the list, that reference will point at the new node, and the link field of the new node will have the value `null`. Look back at Figures 14.4, 14.5, and 14.6 provided above to see how the link field in the last node in the list always has the value `null`.

Notice that the only named reference within the linked list is the reference `myHead`, that points to the first node in the list. In order to access the nodes in the list, we will always have to start with the reference `myHead`.

Exercise 14.2: Draw a picture of the linked list structure `myStringLinkedList` right after the third node containing a reference to `"Barbara"` is added to the list. This picture shows the list before `"Betty"` and `"Mikey"` have been added.

The Basics of a Linked List in Java

In order to construct a linked list in Java, we will need two classes: one for the list itself and one for the nodes. We can then declare and construct variables of the linked list class. We will begin by introducing the `Node` class. As noted earlier, an object of class `Node` has two fields, a data field and a link field. These fields correspond to instance variables in Java. To make our design reusable, we will assume the data field is of the generic type `E`, indicating that our linked list can hold any class of object that Java provides or that we define. In the discussion below we will use the terminology that "a `Node` contains a reference to a data item".

Hence, the beginning of the `Node` class looks as follows:

```
public class Node<E> {
        private E myData;
        private Node<E> myLink;

        ▪▪▪ //more to come
} //Node class
```

Like most classes, the Node class will have at least one constructor. Normally when we construct a Node, we will want to pass in the data that the Node will reference. Hence, the constructor would look as follows if we assign the default value of null to the link field:

```
public Node (E theData) {
        myData = theData;
        myLink = null;
} // 1 parameter constructor
```

It may also be useful to write a two parameter constructor for the Node class, where both the data and the link fields are passed in as parameters to the constructor. This method is left to you as an exercise.

Exercise 14.3: Write the two parameter constructor for the Node class.

The Node class will also need get and set methods for each instance variable. These are like other standard get and set methods, and therefore are left as an exercise.

Exercise 14.4: Write the following methods for the Node class:
a) setData
b) getData
c) setLink
d) getLink

When you have completed the above exercises, the Node class is complete.

The second class for the list we will name WCSLinkedList. We will think of objects of this class as linked lists of E objects, and hence use the type parameter <E> in the class header. Referring to the pictures above, we can see that the WCSLinkedList class has one instance variable, which we have named myHead, that can point or refer to a Node<E> object. Hence, the class of the instance variable myHead is Node<E>. In constructing the initial list, we indicate the list is empty by assigning myHead the value null. The declaration of the instance variables and the constructor are shown below.

```
public class WCSLinkedList<E> {

    private Node<E> myHead;//refers to the first node in the list

    public WCSLinkedList() {
            myHead = null;
    }// 0-parameter constructor

    ■■■ //more to come
}  //WCSLinkedList class
```

Using our definitions so far, we can declare and construct the simple list structure shown in Figure 14.2.

Adding Elements by Brute Force

How do we add to a linked list? To continue our discussion of linked list syntax and semantics, we will begin by adding to the list in the most basic way possible. All of the statements provided in this section would be in the `WCSLinkedList` class itself, in its `main` method simply for testing purposes. In general, we would never want to construct lists in this way. However, using what is presented here, we will be able to create algorithms that generalize the process.

An empty list object could be declared and constructed within our `main` method using the following statement:

```
public static void main (String args []){
    WCSLinkedList<String> myStringLinkedList = new WCSLinkedList< >();
}//main
```

The identifier `myStringLinkedList` is the identifier used to reference the list object from the code where it is constructed. The `private` instance variable `myHead` is crucial in that it points to the first node in the linked list of nodes. In the discussion below, we take the perspective from *within* the class `WCSLinkedList` and refer to our views of the list from that perspective as "internal" views of the list. Consider the following internal view of the empty list:

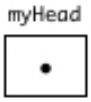


Figure 14.7 Internal view of `myStringLinkedList` when the list is empty

Putting the first element in this list requires constructing a new `Node`, and assigning the reference `myHead` to point to it. This is a task you have performed many times, and is accomplished with the highlighted line of code within the `main` method of the `WCSLinkedList` class:

```
public static void main (String args []){
        WCSLinkedList<String> myStringLinkedList = new WCSLinkedList< >();
        myStingLinkedList.myHead = new Node<String>("George");
}//main
```

The diamond operator of Java 1.7 can be used once again to shorten the highlighted code above. The type parameter needed for the Node construction can be inferred by the compiler from the type of `myStringLinkedList.myHead` which is `Node<String>`. Hence, we use the shorter version of the code displayed below.

```
public static void main (String args []){
        WCSLinkedList<String> myStringLinkedList = new WCSLinkedList< >();
        myStringLinkedList.myHead = new Node< >("George");
}//main
```

Note that we are using the one parameter Node constructor, that expects an argument of type String. Since "George" is a literal String value, it is an acceptable argument. Once this is accomplished, the internal view of the list is as follows:

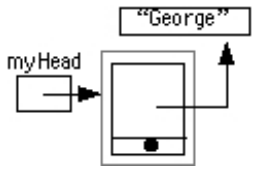


Figure 14.8 Internal view of myStringLinkedList after one Node has been added

Putting the next Node in the list is a bit trickier. Like adding the initial element to the list, we must construct a Node object, and then have a reference point to it. Which reference do we want to point to the new Node? We want the new Node to be pointed to by the link field in the Node that contains the reference to "George". The reference to that Node is myHead. To change the value of one of the Node's instance variables, we need to use a modifier or set method, in this case, the method setLink. The following highlighted line could be used within the main method of the WCSLinkedList class:

```
public static void main (String args []){
        WCSLinkedList<String> myStringLinkedList = new WCSLinkedList< >();
        myStringLinkedList.myHead = new Node< >("George");
        myStringLinkedList.myHead.setLink(new Node< >("Lydia"));
}//main
```

Once this is accomplished, the internal view of the list is as in Figure 14.9 below:

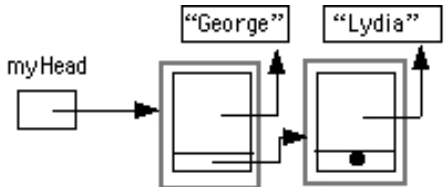


Figure 14.9 Internal view of myStringLinkedList after two Nodes have been added

Putting the third Node in the list again requires constructing a Node object and making the appropriate reference point to it. In this case, the reference that should point to it is the link reference found in the Node that contains the reference to Lydia. The construction is again trivial, but setting the link becomes more difficult, because access to the link must begin with the reference myHead.

Note the following:

- myHead is the reference to the Node containing the reference to "George".
- myHead was given an initial value using an assignment statement.
- myHead.setLink(…) modifies the value of the link field in the Node that contains the reference to "George". In our example, it changed its value from null to a reference to the Node that contains the reference to "Lydia".
- myHead.getLink() returns the value of the link field in the Node that contains the reference to "George" so that we can access the Node to which it points, in this case, the Node that contains the reference to "Lydia".

- Since `myHead.getLink()` is a reference to the `Node` that contains the reference to `"Lydia"`, we can use it *in conjunction with* `get` and `set` methods to access or modify the link field of the `Node` that contains the reference to `"Lydia"`.

Hence, to add a new `Node` to the end of our current list, we would use the final line of code within the `main` method of the `WCSLinkedList` class:

```
public static void main (String args []){
    WCSLinkedList<String> myStringLinkedList = new WCSLinkedList< >();
    myStringLinkedList.myHead = new Node< >("George");
    myStringLinkedList.myHead.setLink(new Node< >("Lydia"));
    myStringLinkedList.myHead.getLink() .setLink(new Node< >("Barbara"));
}//main
```

Note, the highlighted portion of the statement is a reference to the `Node` that contains the reference to `"Lydia"`. We are then requesting that the field `myLink` of that `Node` be made to point at a new `Node` containing the reference to `"Barbara"`.

The internal view of the list now is as in Figure 14.10:

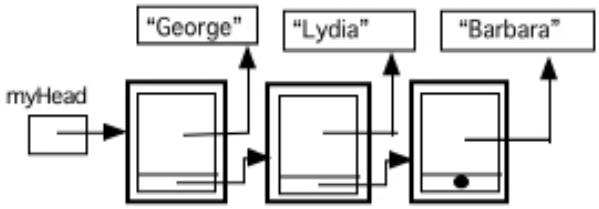


Figure 14.10 Internal view of `myStringLinkedList` after three `Nodes` have been added

As we attempt to add the fourth node to the list, we will annotate the picture, along with the statement we need:

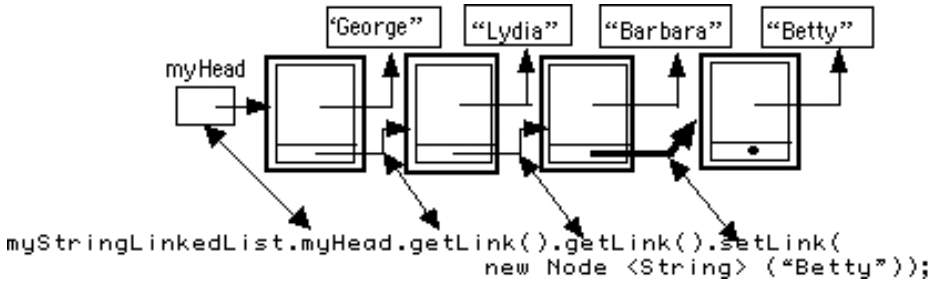


Figure 14.11 An annotated statement to add "Betty" to the list. The thick arrow indicates the link being modified.

```
public static void main (String args []){
    WCSLinkedList<String> myStringLinkedList = new WCSLinkedList< >();
    myStringLinkedList.myHead = new Node< >("George");
    myStringLinkedList.myHead.setLink(new Node< >("Lydia"));
    myStringLinkedList.myHead.getLink().setLink(new Node< >("Barbara"));
    myStringLinkedList.myHead.getLink().getLink().setLink(
                                        new Node< >("Betty"));
}//main
```

Exercise 14.5: Complete the list by writing the statement needed to add "Mikey" to the end of the list.

The technique we have used thus far to add to the list is tedious, and becomes more so as the list gets longer. Moreover, it requires us to know how many Nodes are in the list each time we wish to add a new Node. Several new techniques for adding to a list will be presented later in this chapter.

Printing List Elements by Brute Force

Now that we have built our linked list, we will play with it a bit more to make sure we understand how to use the methods provided by the Node class to manipulate the list. Internally, the list now looks as follows:

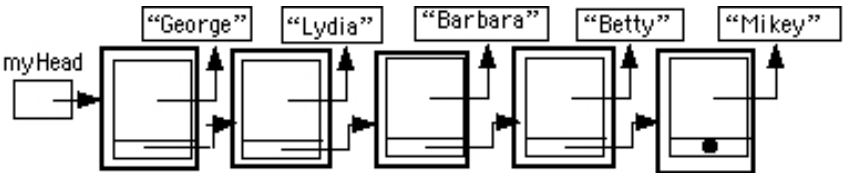


Figure 14.12 Internal view of the five element linked list.

How do we output the data in the first Node? The data is stored in the field myData of the Node referenced by myHead. Hence, we need to use the getData method to access that data. The highlighted statement below written in the main method of class WCSLinkedList outputs the data in the first Node, the String "George", to the Java Console view. Note, we are assuming that the object that has the reference to the data in question has a toString method that returns a String representation of the data in an acceptable manner.

```
public static void main (String args []){
    WCSLinkedList<String> myStringLinkedList = new WCSLinkedList< >();
    myStringLinkedList.myHead = new Node< >("George");
    myStringLinkedList.myHead.setLink(new Node< >("Lydia"));
    myStringLinkedList.myHead.getLink().setLink(new Node< >("Barbara"));
    myStringLinkedList.myHead.getLink().getLink().setLink(
                                                new Node< >("Betty"));
    System.out.println(myStringLinkedList.myHead.getData().toString());
}//main
```

Now, how do we output the second piece of data? We must first get to the link that refers to it, then we must use the getData method to access the data field of the Node. The link that refers to the Node that has the reference to "Lydia" is the myLink field in the Node referred to by myHead. Hence, we need to get that link. The highlighted last line of the main method will output "Lydia".

```
public static void main (String args []){
    WCSLinkedList<String> myStringLinkedList = new WCSLinkedList< >();
```

```
    myStringLinkedList.myHead = new Node< >("George");
    myStringLinkedList.myHead.setLink(new Node< >("Lydia"));
    myStringLinkedList.myHead.getLink().setLink(new Node< >("Barbara"));
    myStringLinkedList.myHead.getLink().getLink().setLink(
                                                new Node< >("Betty"));
    System.out.println(myStringLinkedList.myHead.getData().toString());
    System.out.println(myStringLinkedList.myHead.getLink().getData().toString());
}//main
```

Now let us skip ahead and attempt to output the data in the last Node, i.e., the Node that has the reference to Mikey. Again, we will use a picture to help us:

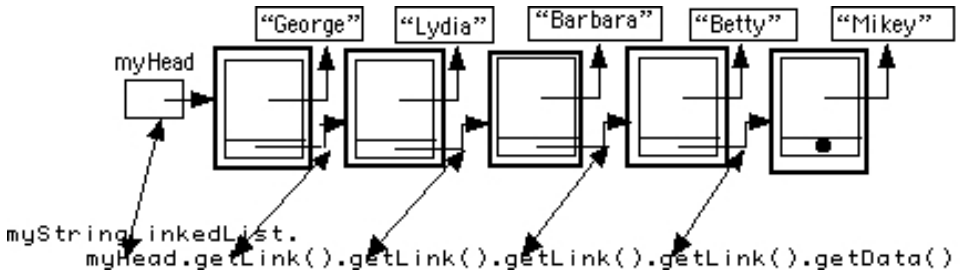


Figure 14.13 How to access the data in the fifth Node of the list

We now need to use the toString method on the returned value and output is produced by the last line of the main method as highlighted below.

```
public static void main (String args []){
    WCSLinkedList<String> myStringLinkedList = new WCSLinkedList< >();
    myStringLinkedList.myHead = new Node< >("George");
    myStringLinkedList.myHead.setLink(new Node< >("Lydia"));
    myStringLinkedList.myHead.getLink().setLink(new Node< >("Barbara"));
    myStringLinkedList.myHead.getLink().getLink().setLink(
                                                new Node< >("Betty"));
    myStringLinkedList.myHead.getLink().getLink().getLink().setLink(
                                                new Node< >("Mikey"));
    System.out.println(myStringLinkedList.myHead.getData().toString());
    System.out.println(myStringLinkedList.myHead.getLink().getData().toString());
    System.out.println(myStringLinkedList.myHead.getLink().getLink().
          getLink().getLink().getData().toString());
}//main
```

Exercise 14.6: Draw an annotated picture like that shown in Figure 14.13 to show the line of code needed to print the data field of the Node containing the reference to "Barbara".

Exercise 14.7: Write the line of code needed to print the data of the `Node` containing the reference to `"Betty"`.

Deleting List Elements by Brute Force

Now we will delete some items from the list. Let us delete the node that contains the reference to `"Lydia"` from the list. To begin with, let us draw a picture that shows the link we want to change.

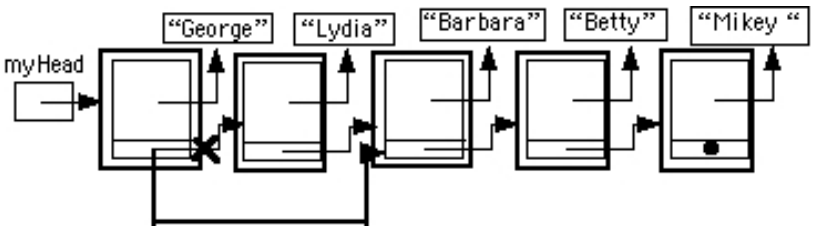


Figure 14.14 How to delete the `Node` that contains the reference to `"Lydia"`

To delete the `Node` that contains the reference to `"Lydia"` from the list, all we need to do is to re–link the `Node` before it so that it bypasses the `Node` that contains the reference to `"Lydia"`. To do this we need to get at two pieces of information from the list. We need to modify the link reference coming out of the `Node` that contains the reference to `"George"`, *and* we need to access the link reference coming out of the `Node` that contains the reference to `"Lydia"`. Why do we need this second value? We need that second reference because we want the link from the `Node` that contains the reference to `"George"` to point to the `Node` to which the `Node` that contains the reference to `"Lydia"` is pointing (i.e., we want the link from the `Node` that contains the reference to `"George"` to point to the `Node` that contains the reference to `"Barbara"`. Wild, isn't it? But you will get used to it!) Let us start by writing the portion of the statement that will change the existing link. The `Node` that contains the reference to `"George"` is referred to by `myHead`. Hence, we begin by writing

```
myStringLinkedList.myHead.setLink(…);
```

Now we must provide the argument to the `setLink` method. Think of the `setLink` as "refer to the same thing as". Hence, we want the link from the `Node` that contains the reference to `"George"` to *point to the same thing as* the `myLink` field of the `Node` that contains the reference to `"Lydia"`. We need to supply the `myLink` field of the `Node` that contains the reference to `"Lydia"`. This is:

```
myStringLinkedList.myHead.getLink().getLink()
```

(Try drawing the picture if you are unsure of this.) Our complete statement is:

```
myStringLinkedList.myHead.setLink(
    myStringLinkedList.myHead.getLink().getLink());
```

Following execution of this statement, our linked list will look as follows in Figure 14.15:

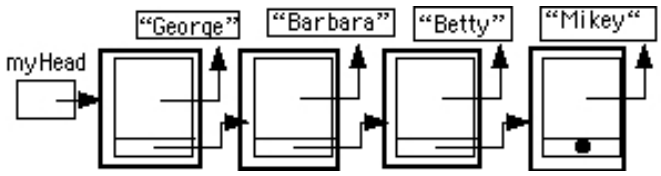


Figure 14.15 The list after deletion of the Node that contains the reference to "Lydia"

It does not matter how many Nodes follow the Node we deleted in the list. Even if there were 100 Nodes after the Node that we wished to delete, the process would still be the same, and no additional processing needs to be done by the computer.

Now let us delete the Node that contains the reference to "Betty". First, we want to modify the reference that is pointing to that Node. That reference is the link field in the Node that contains the reference to "Barbara". Hence, we could write the following code:

```
myStringLinkedList.myHead.getLink().setLink(…);
```

We want to modify this reference to make it point to the same Node to which the Node that contains the reference to "Betty" is pointing (i.e., to the Node that contains the reference to "Mikey"). That would be myStringLinkedList.myHead.getLink().getLink().getLink(). Hence, our complete statement is:

```
myStringLinkedList.myHead.getLink().setLink(
    myStringLinkedList.myHead.getLink().getLink().getLink());
```

Now the list looks as follows:

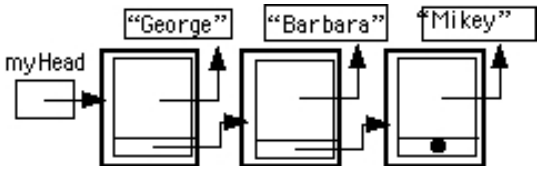


Figure 14.16 The list after deletion of the Node that contains the reference to "Betty"

In all of these cases we have been deleting an element from the middle of list. Now let us look at how to delete the first and last items from the list. First let us delete the last item from the list, the Node that contains the reference to "Mikey". A diagram of the situation is given below:

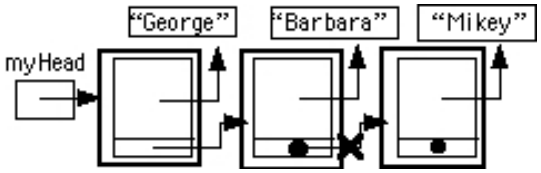


Figure 14.17 The process of the deletion of the Node that contains the reference to "Mikey"

All we need to do is assign null to the link from the Node that contains the reference to "Barbara". This is easily done with the following statement:

```
myStringLinkedList.myHead.getLink().setLink(null);
```

Our list now looks as follows in Figure 14.18:

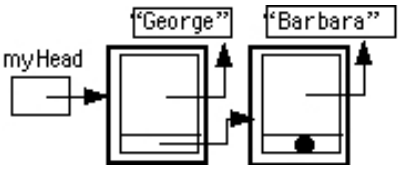


Figure 14.18 The list after deletion of the Node that contains the reference to "Mikey"

Finally, we will delete the Node that contains the reference to "George" from the list. The process is described in the diagram below:

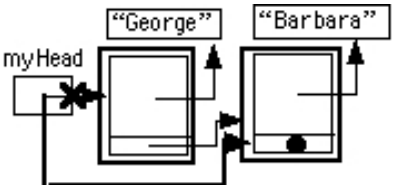


Figure 14.19 The process of the deletion of the Node that contains the reference to "George"

In this case we are *not* modifying a link field of a Node. Instead, we are making the reference myHead point to a different Node. This is done with a simple assignment statement. The assignment operator in this case can also be read as "refers to the same thing as". Hence, what appears on the right–hand side of the assignment operator must be a reference already pointing at the object to which we wish myHead to point. That would be myStringLinkedList.myHead.getLink().

```
myStringLinkedList.myHead = myStringLinkedList.myHead.getLink();
```

This line of code effectively deletes the Node that contains the reference to "George" from the list, leaving only the Node that contains the reference to "Barbara" in the list.

Notice that any Node that does not have a reference pointing to it is essentially deleted—there is no way to access it. The JRE performs a process called Garbage Collection that looks for these "lost in space" memory locations and reclaims the memory they occupy. It is, therefore, very important never to reassign myHead from the Node at the front of the list, unless you are intentionally deleting items from the front of the list, emptying the entire list, or replacing the entire list.

We have assembled the main method of the WCSLinkedList class where all the additions and deletions from the list were performed below.

```
public static void main(String args[]){
    WCSLinkedList<String> myStringLinkedList = new WCSLinkedList< > ();
    myStringLinkedList.myHead = new Node< >("George");
    myStringLinkedList.myHead.setLink(new Node< >("Lydia"));
    myStringLinkedList.myHead.getLink().setLink(new Node< >("Barbara"));
    myStringLinkedList.myHead.getLink().getLink().setLink(
            new Node< >("Betty"));
    myStringLinkedList.myHead.getLink().getLink().getLink().setLink(
            new Node< >("Mikey"));
```

```
        System.out.println(myStringLinkedList.myHead.getData().toString());
        System.out.println(myStringLinkedList.myHead.getLink().getData().toString());
        System.out.println(myStringLinkedList.myHead.getLink().getLink().
                    getLink().getLink().getData().toString());
        myStringLinkedList.myHead.setLink(
                    myStringLinkedList.myHead.getLink().getLink());
        myStringLinkedList.myHead.getLink().setLink(
                    myStringLinkedList.myHead.getLink().getLink().getLink());
        myStringLinkedList.myHead.getLink().setLink(null);
        myStringLinkedList.myHead = myStringLinkedList.myHead.getLink();
    }//main
```

Exercise 14.8: Assume the initial internal list is as shown in Figure 14.20 below. Write the code to delete the `Node` that contains the reference to `"Mikey"` from the list.

Exercise 14.9: Assume the initial internal list is as shown in Figure 14.20 below. Write the code to delete the `Node` that contains the reference to `"Betty"` from the list.

Exercise 14.10: Assume the initial internal list is as shown in Figure 14.20 below. Write the code to delete the `Node` that contains the reference to `"Barbara"` from the list.

Exercise 14.11: Assume the initial internal list is as shown in Figure 14.20 below. Using one statement, delete the `Node`s that contain the references to `"Lydia"` and `"Barbara"` respectively from the list.

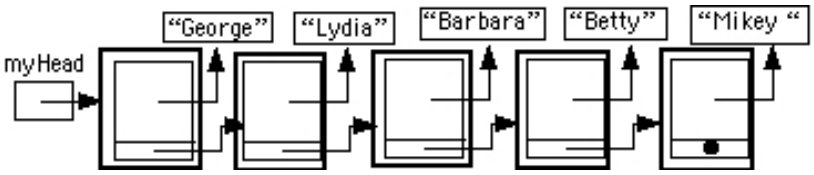


Figure 14.20 A five element linked list.

Inserting List Elements by Brute Force

Now we will do a few examples of modifying a list by adding elements in various places. Let us assume we have our five element linked list as in Figure 14.20 to start.

We would like to insert a `Node` that contains a reference to `"Leilani"` between the `Node` that contains a reference to `"Lydia"` and the `Node` that contains a reference to `"Barbara"`. This means we wish to redraw the diagram as follows:

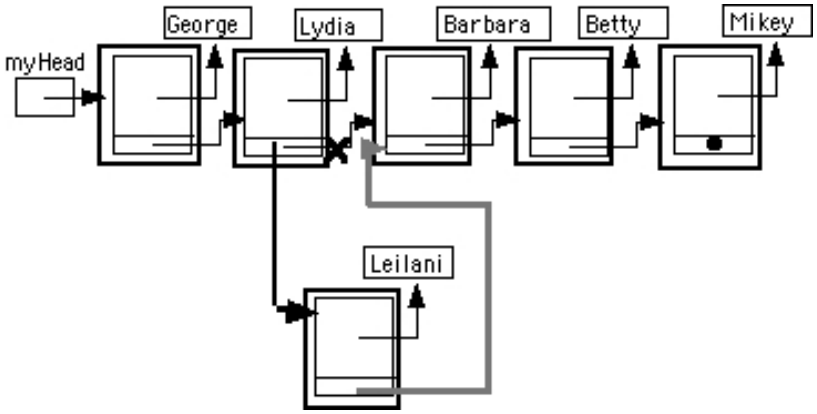


Figure 14.21 Inserting a new Node between existing Nodes.

Inserting the new Node into the list requires changing the link in the Node that will precede the new Node, and making the new Node point to the object that will succeed it in the list. Additionally, we actually need to construct the new Node. For clarity, we will first show you how to insert the Node in the list after it has been constructed. The initial situation is given in Figure 14.22.

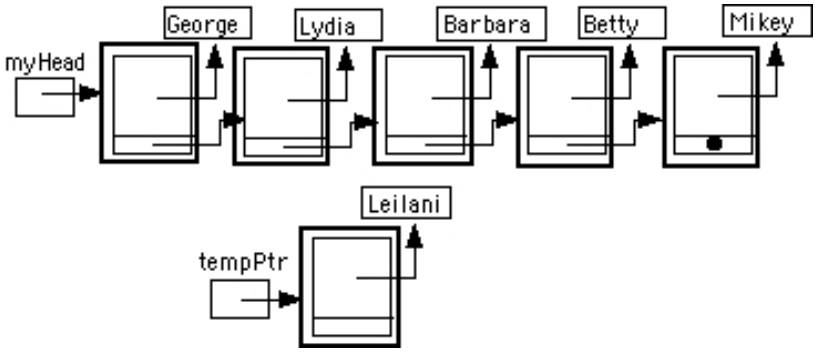


Figure 14.22 Inserting a new Node between existing Nodes.
Just after the new Node has been constructed.

Notice that in order to construct the new Node before inserting it, we have had to set up a temporary pointer to it. This is done as follows:

```
Node<String> tempPtr = new Node< >("Leilani");
```

Now, looking back at Figure 14.21, we see that two links need to be reset. The question arises: does it matter which link we reset first? Consider the earlier statements that a Node with nothing pointing to it is essentially deleted from the list. This provides us to the answer to our question. If we reassign the pointer from the Node that contains the reference to Lydia, the situation looks as follows in Figure 14.23:

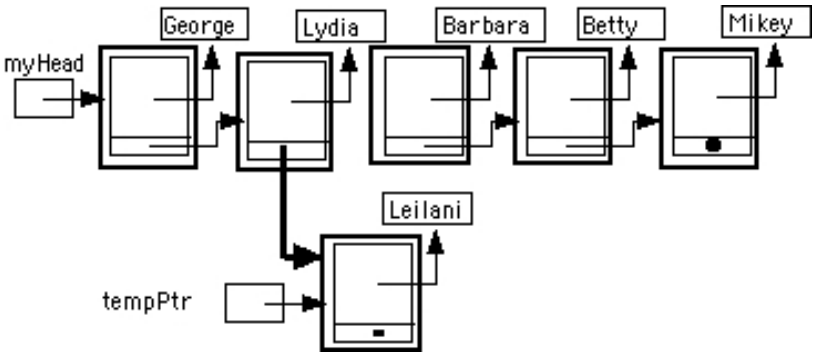


Figure 14.23 Situation if we reassign the myLink field of the Node that contains the reference to "Lydia" first.

Notice that there is no longer a reference to the Node that contains the reference to "Barbara". Not only have we lost that Node, but we have also lost the Nodes that come after it in the list, because we need to have the Node that contains the reference to "Barbara" to access them. Hence, we must assign the myLink field of our temporary Node so that it refers to the Node containing the reference to "Barbara" first. This is done with the statement:

```
tempPtr.setLink(myStringLinkedList.myHead.getLink().getLink());
```

Notice that we have two getLinks after myHead, because we want the reference that is stored in the second Node. This statement draws the grey line shown in Figure 14.21.

We can now assign the link reference in the Node that contains the reference to "Lydia" so that it refers to the new Node, which can also be referred to as the Node referred to by tempPtr. This is done with the following statement:

```
myStringLinkedList.myHead.getLink().setLink(tempPtr);
```

Notice that we are resetting the link in the second Node. Hence, we have two method invocations concerning links, a getLink followed by a setLink. This line of code draws the thick black line shown in Figure 14.21.

Exercise 14.12: Assume the initial list is as shown in Figure 14.20. Write the code to add a Node with your name between the Node that contains the reference to "Barbara" and the Node that contains the reference to "Betty".

Exercise 14.13: Assume the initial list is as shown in Figure 14.20. Write the code to insert a Node that contains a reference to "Pooh Bear" at the front of the list (so that it becomes the first Node in the list.)

Although we used three statements to accomplish our task of insertion, we could have done it all in one statement, using our two parameter Node constructor. The following line of code also inserts the Node that refers to Leilani in the list as described above:

```
myStringLinkedList.myHead.getLink().setLink(
    new Node< >("Leilani",myStringLinkedList.myHead.getLink().getLink()));
```

A toString Method for a Linked List

Before we go on to build lists in other ways, it is useful to look at how we write the `toString` method that will allow us to print the contents of the list. Recall that when our underlying representation was an array, our `toString` method looked as follows:

```
public String toString() {
    String toReturn = "";
    for (int i = 0; i < myCount; i++) {
        toReturn += myArray[i] +"\n";
    }// for
    return toReturn;
}  //toString
```

The variable `i` moves us through the array:

- `i` gets the initial value `0`, starting us at the beginning of the array.
- we check that `i` does not exceed the number of elements we have in the array
- we process the element at position `i`
- we increment `i` to move to the next element in the array.

We need a similar way to move through the linked list. Because we do not know the specific size of the linked list, we will use a `while` loop, rather than a `for` loop to move through the linked list.

We will have a reference, named `mover`, of class `Node<E>`, that moves us through the linked list:

- `mover` will initially refer to the beginning of the list. We do this by assigning it to point to the `Node` to which `myHead` is pointing.
- We check that `mover` has not reached the end of the list. We will know we are at the end of the list when `mover` is `null`.
- We process the element to which `mover` is referring.
- We move `mover` along to the next element of the list.

This algorithm translates into the following code:

```
public String toString() {
    String toReturn = "";
    Node<E> mover = myHead;
    while (mover != null) {
        toReturn += mover.getData() + "\n";
        mover = mover.getLink();
    }// while
    return toReturn;
}  //toString
```

We must have a temporary reference like `mover`, because if we use our only existing reference, `myHead`, we will lose `Node`s as we move it through the list. Also recall that in the line of code

```
toReturn += mover.getData() + "\n";
```

the `toString` method is implicitly invoked on `mover.getData()` since it is being used in a `String` concatenation operation.

Although this method is specifically designed to build a `String`, it has general properties that are useful whenever we want to traverse an entire linked list. These parts are highlighted below:

```
public String toString() {
    String toReturn = "";
    Node<E> mover = myHead;
    while (mover != null) {
        toReturn += mover.getData() + "\n";
        mover = mover.getLink();
    }//while
    return toReturn;
}  //toString
```

The first highlighted line, `Node<E> mover = myHead;`, creates a reference to move through the list, and starts it at the beginning of the list. The line `while (mover != null) {` makes sure that we have not gone beyond the end of the list. Additional expressions can be added to this condition, but the first condition should always be the check to make sure `mover` has not moved past the end of the list. For example, if we were searching the list for a particular piece of data, the condition might look as follows:

```
while (mover != null && !mover.getData().equals(findMe))
```

It is important that the condition `mover != null` come first. If `mover` is `null`, and we try to invoke a method, such as `getData()` the JVM will throw a `Null Pointer Exception`—a run-time error.

The third highlighted line, `mover = mover.getLink();`, has the effect of moving our reference `mover` to the next `Node` in the linked list.

Methods to Add to a Linked List

We have now extensively practiced with `getLink` and `setLink`. It is time to move on to write more general methods that allow us to add new `Node`s to a linked list. They are three common ways to build a linked list:

- Add to the front of the list.
- Add to the end of the list.
- Add to the list based on some ordering scheme, in other words, build a sorted list.

When we created lists using the underlying array representation, we built FIFO–like (First In, First Out) lists. The first item we put in the list was the first item we saw when we printed out the list. An analogous situation is the line at a supermarket. The first person to enter the line is the first person to be waited upon, the second person to enter

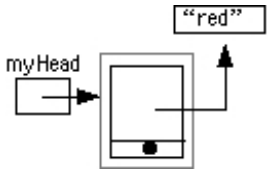
the line is the second person to be waited upon, and so on. FIFO lists are also called *queues*. In fact, in England, people queue up to get into concerts, rather than line up. If we never sort the list and always add to the end and remove from the beginning, we have a pure FIFO list.

LIFO (Last In, First Out) lists are also referred to as *stack*s. With a LIFO list, the last item entered in the list is the first item to be processed. An analogous situation is a stack of plates in a cupboard. The first plate you put in goes at the bottom of the stack. The next plate goes on top of that, and so on. When we go to remove an plate from the cupboard, the last plate we put on top of the stack will be the first plate that we remove. (Which is a good thing if you don't want to break them all.) Stacks are heavily used by compilers and runtime systems, and you will encounter them often in future Computer Science courses.

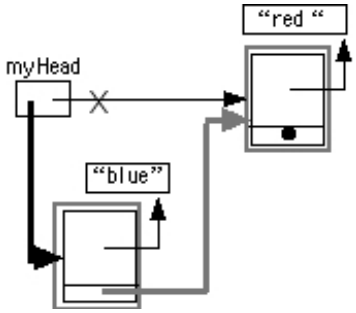
Adding at the Front of a List

The first method we write to add items to a linked list will be a LIFO method, because it is the easiest. Consider the four pictures below, showing three `String`s being added to build a LIFO list.

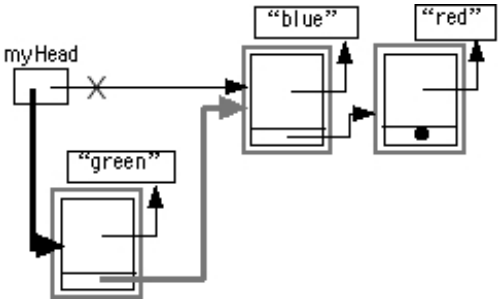
(a) The list before insertion.



(b) The list after the first item "red" has been inserted.



(c) The list as the second item "blue" is being inserted.



(d) The list as the third item "green" is being inserted.

Figure 14.24 Building a LIFO list.

Notice that the same links are redrawn in Figure 14.24 (c) and (d). The same links will be redrawn whenever we add a new item to the list.

Exercise 14.14: Draw a diagram like that shown in Figure 14.24(d) above that shows the `String` `"purple"` being added to the list. You should assume the list to which your are adding is the list after the relinking shown in Figure 14.24(d) has been completed.

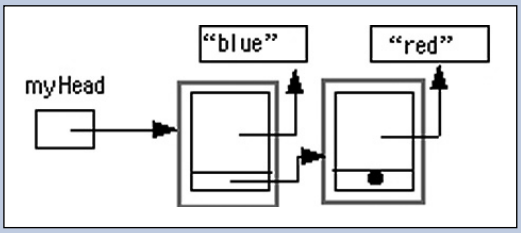
The links that are redrawn in Figure 14.24(c) and (d) are very similar to those drawn when we earlier inserted an item into the list. In fact, we can approach the insertion process using the same algorithm:

1) Construct a new `Node` pointed to by `tempPtr`.
2) Assign the `myLink` field of the `Node` referred to by `tempPtr` so that it refers to the `Node` to which `myHead` is referring.
3) Assign `myHead` so that it refers to the `Node` to which `tempPtr` is referring.

We need to incorporate these steps into a method. We will name the method `addToFront`. Like most methods that add to a list, it will have as a parameter the data to be stored in the `Node` in the form of a reference to an object of generic type `E`. The method that we could add to our `WCSLinkedListclass` is as follows:

```
public void addToFront (E theData) {
    Node<E> tempPtr = new Node< >(theData);
    tempPtr.setLink(myHead);
    myHead = tempPtr;
} // addToFront
```

Exercise 14.15: Trace the following incorrect version of the `addToFront` method by modifying the picture below. Cross out the original links and redraw the links as necessary. Start with the diagram shown below. Assume `theData` is the `String` `"white"`.




```
public void addToFront (E theData) {
    Node<E> tempPtr = new Node< >(theData);
    myHead = tempPtr;
    tempPtr.setLink(myHead);
}// addToFront – incorrect
```

Adding at the End of a List

Building a FIFO list requires adding each new Node at the end of the list and is a little more difficult than adding at the head of the list. There are two common methods for finding the end of the list so that we can add a Node there.

- Move a temporary pointer through the list to find the last Node.
- Maintain a pointer to the last Node of the list.

Version 1, Adding to the end of a list by locating the last Node each time

To find the end of the list with a temporary reference, we must begin at the head of the list and move the reference along until we reach the last Node. This is similar to what we did when traversing the list in the toString method—except that we want to stop at the last Node, not when we have gone past the last Node. Additionally, we must have a special case for inserting the first Node in the list. We can recognize this case because the reference myHead will have the value null. We will deal with that case first. For the second case, we will adapt the while loop used above in the toString method to look ahead of the Node to which mover is referring:

```
public void addAtTail1(E theData) {
     if (myHead == null) {
          myHead = new Node< >(theData);
     } else {
          Node<E> mover = myHead;
          while (mover.getLink() != null) {
               mover = mover.getLink();
          }  //while

               ■■■ //more to come
     }  //else
}  // addAtTail1
```

Notice that the looping condition does not involve mover, but instead mover.getLink(). This is what makes the loop look ahead, and stop at the last Node, rather than going beyond the last Node.

Now that we have found the last Node in the list, we need to put the new Node in the list. The situation is as shown in Figure 14.25:

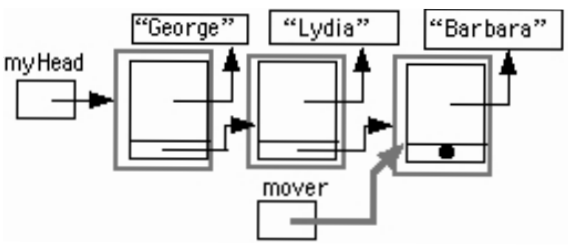


Figure 14.25 The situation after we have positioned mover pointing to the last Node in the list.

We can now construct the new Node making the pointer in the Node that refers to "Barbara" point to it. This is done with the following line of code:

```
    mover.setLink(new Node< >(theData));
```

The completed method is as follows:

```
public void addAtTail1 (E theData) {
     if (myHead == null) {
          myHead = new Node< >(theData);
     } else {
          Node<E> mover = myHead;
          while (mover.getLink() != null) {
               mover = mover.getLink();
          }  //while
          mover.setLink(new Node< >(theData));
     }  //else
}  // addAtTail1
```

This method is useful for small lists, but the larger the list becomes, the more work the computer must do to add an element to the end of the list. For example, if the list contains 1000 elements, 999 comparisons must be done by the `while` loop before the last `Node` in the list is found. The next method addresses this problem.

Version 2, Adding to the end of a list by maintaining a pointer to the end of the list

In order to maintain a reference to the end of the list, we first need an additional instance variable in the `WCSLinkedList` class. We will name that instance variable `myTail`. Clearly, in an empty list, both `myHead` and `myTail` will be `null`. In a list that contains only one `Node`, both `myHead` and `myTail` refer to that single `Node`. In general, when we add a `Node` to the end of a list, `myTail` changes much more often than `myHead`. If you only add `Nodes` to the end of a list, `myHead` only changes when the list changes from an empty list to a list containing one `Node`.

Let us first change the code of the `WCSLinkedList` class by adding a new instance variable named `myTail` and adding a line of code to the constructor that initializes `myTail`.

```
public class WCSLinkedList<E> {
     private Node<E> myHead;
     private Node<E> myTail;

     public WCSLinkedList() {
          myHead = null;
          myTail = null;
     }// 0-parameter constructor

     ■■■ //more to be written
}  //WCSLinkedList class
```

Exercise 14.16: Write a method named addAtTail2 for the WCSLinkedList class. addAtTail2 will have one parameter of generic type E named theData. addAtTail2 will construct a Node with theData and appropriately change myHead, myTail, and any other links in the list to affect the addition of the new Node at the tail of the WCSLinkedList.

Methods to Delete from a Linked List

They are three common ways to delete from a linked list:

- Delete the element at the head of the list.
- Delete the element at the tail of the list.
- Delete an element from a sorted list.

We leave the coding of the first two ways as exercises in this chapter and the coding of the third way as an exercise in the next chapter.

Exercise 14.17: Write a boolean-valued method named deleteAtHead for the WCSLinkedList class. deleteAtHead will have no parameters. deleteAtHead will delete the Node at the head of the list as long as the list is not empty and return true. If the list is empty deleteAtHead will return false. deleteAtHead should *not* throw an exception in the case of an attempt to delete from an empty list. deleteAtHead will appropriately change myHead, myTail, and any other links in the list to affect the deletion of the Node at the head of the WCSLinkedList.

Exercise 14.18: Write a method named deleteAtTail for the WCSLinkedList class. deleteAtTail will have no parameters. deleteAtTail will delete the Node at the tail of the list as long as the list is not empty and return true. If the list is empty deleteAtTail will return false. deleteAtTail should not throw an exception in the case of an attempt to delete from an empty list. deleteAtTail will appropriately change myHead, myTail, and any other links in the list to effect the deletion of the Node at the tail of the WCSLinkedList.

Exercise 14.19: Write a class named WCSStackLL<E> that implements a stack with a linked list. Refer to Exercise 12.6 for descriptions of the required methods and the class StackTest to test your class WCSStackLL<E>.

Exercise 14.20: Write a class named WCSQueueLL<E> that implements a queue with a linked list. Refer to Exercise 12.7 for descriptions of the required methods and the class QueueTest to test your class WCSQueueLL<E>.

Exercise 14.21: Complete the class named `WCSLinkedList<E extends Comparable<E>>` that implements a linked list of items that can be ordered.

Your `WCSLinkedList <E extends Comparable<E>>` class should have at least three private instance variables. The first instance variable is of class `Node<E>` and is a reference to the head of the list. The second instance variable is of class `Comparator<E>` and might be used to rebuild the list in an order specified by the `Comparator`. The third instance variable keeps a count of items in the list.

One constructor of class `WCSLinkedList <E extends Comparable<E>>` will have one parameter that is a reference to a `Comparator<E>` object. The other constructor will have no parameters.

Your class `WCSLinkedList <E extends Comparable<E>>` should have at least the following instance methods:

```
public boolean add(E addMe)
public E search(E findMe)
public boolean delete(E deleteMe)
public boolean isEmpty()
public int getCount()
private int myCompare(E firstItem, E secondItem)
```

`public void sort()` (Note that you will need to copy all items into an array, sort the array, and build a new linked list by adding items from the sorted array.)

```
public String saveToFile(String fileName)
public String loadFromFile(String fileName)
public String toString()
```

You will need a `Node<E>` class to work in conjunction with your `WCSLinkedList <E extends Comparable<E>>` class.

Exercise 14.22: Continuation of Exercise 13.1

a) Revise your `Database` class so that you declare, construct, and use two `WCSLinkedList <AddressInfo>` objects, one that can be ordered alphabetically and one that can be ordered by address. When you construct your `WCSLinkedList<AddressInfo>` that can be ordered by address, you should pass the public reference to the `AddressComp` object to the constructor.

b) When you add a new record to the database, you add it to both `WCSLinkedList< AddressInfo>` objects.

c) When you delete a record from the database, you delete it from both `WCSLinkedList< AddressInfo>` objects.

d) Your alphabetical display button needs to rebuild the `WCSLinkedList <AddressInfo>` that is ordered alphabetically and display the list.

e) Your address display button needs to rebuild `WCSLinkedList <AddressInfo>` that is ordered by address and display the list.

f) When your code saves the contents of the database to a file, save only the items in the alphabetical list.

g) When your code loads items from a file, add each item to both lists.

Exercise 14.23: Continuation of Exercise 13.2

a) Revise your `Database` class so that you declare, construct, and use two `WCSLinkedList <BirthInfo>` objects, one that can be ordered alphabetically and one that can be ordered chronologically. When you construct your `WCSLinkedList<BirthInfo>` object that can be ordered chronologically, you should pass the public reference to the `ChronComp` object to the constructor.

b) When you add a new record to the database, you add it to both `WCSLinkedList <BirthInfo>` objects.

c) When you delete a record from the database, you delete it from both `WCSLinkedList <BirthInfo>` objects.

d) Your alphabetical display button needs to rebuild the `WCSLinkedList <BirthInfo>` that is ordered alphabetically and display the list.

e) Your chronological display button needs to rebuild the `WCSLinkedList <BirthInfo>` that is ordered chronologically and display the list.

f) When your code saves the contents of the database to a file, save only the items in the alphabetical list.

g) When your code loads items from a file, add each item to both lists.

Lab: Linked List Lab 1 • Chapter 14

The objectives of this lab are:

- to get practice using the getLink and setLink methods of the Node<E> class
- to get practice writing methods that are similar to the toString method

This lab uses .java files in a folder entitled for_linked_list_lab1

1) Get a copy of the folder named for_linked_list_lab1 onto the **Desktop**.
2) Launch Eclipse.
3) Create a new Java project named with LinkedListLab1 *and* your name.
4) Drag the folder for_linked_list_lab1 from the **Desktop** onto the image of the your LinkedListLab1 folder in the Package Explorer view.
5) Double click on your LinkedListLab1 folder in the Package Explorer view to open the folder.
6) Double click on the for_linked_list_lab1 package in the Package Explorer view to open the folder. You should see that the files TestWCSLinkedList.java, WCSLinkedList.java and Node.java in the for_linked_list_lab1 package.

For steps 7 through 14 below, you will be modifying the main method in the TestWCSLinkedList class. In each of steps 7 through 14, write the necessary code to be inserted into the main method. Unless otherwise noted:

- You should use **only** the methods getLink, setLink, and toString of the Node class, the method toString of the WCSLinkedList class, and the assignment statement.
- You should **not** use iteration unless it is indicated in the statement of the problem.
- You should **not** modify the list unless specifically asked to do so.
- When you are asked to display the list, use System.out.println.

7) Construct a linked list with the strings "B", "G", "K" (in alphabetical order). Display the list. You can do this by invoking the toString method of the WCSLinkedList class.
8) Insert Nodes containing references to "D" and "J" into the list so that the list maintains its alphabetical order. Display the list. *The easiest way to do this is first to create a temporary reference that initially points to the new* Node*. Then insert that* Node *into the list.*
9) Again, maintaining alphabetical order, insert a Node containing a reference to "A" and a Node containing a reference to "Z" in the list. Display the list.
10) Display the first three Nodes in the list.
11) Remove the Node containing the reference to "D" from the list. Display the list.
12) Remove the Node containing the reference to "A" from the list. Display the list.

13) Remove the `Node` containing the reference to "Z" from the list. Display the list.

14) Using just one statement delete the `node`s containing "G" and "J" from the list. Display the list.

For each of steps 15 and 16, you are to write a method in class `WCSLinkedList`. In order to test those methods, you will write code in the `main` method of class `TestWCSLinkedList`. Your test code should initialize the list with the following letters: "A", "B", "E", "R", "T" before testing your methods.

15) Write a method `toStringEveryOther`, that builds up a `String` of every other element in the list, beginning with the first element. The method, like the `toString` method, must use a temporary reference and a `while` loop to move through the list. Invoke the method `toStringEveryOther` in conjunction with `System.out.println` in the `main` method.

16) Write a method `toStringGrtThan`, that takes a single `String` argument. The method should build up a `String` of all the elements in the list that are alphabetically beyond the parameter. The method, like the `toString` method, must use a temporary reference and a `while` loop to move through the list. Invoke the method `toStringGrtThan` in conjunction with `System.out.println` in the `main` method several times to show that it works in a variety of situations.

15

Ordered Linked Lists and Recursive List Processing

INTRODUCTION TO ORDERED LINKED LISTS

In Chapter 14, we introduced the concept of a linked list. We discussed adding and deleting nodes at the head of the list to implement the LIFO data structure known as a stack. We also discussed adding nodes to the tail of the list, allowing us to implement the FIFO data structure known as a queue.

In this chapter, we will discuss ordered linked lists. In other words, the nodes are placed in the list according to some agreed upon ordering scheme that determines when one node's data is "before" another node's data.

Our goal in this chapter is to develop a class `WCSOrderedLinkedList<E extends Comparable<E>>`. Like the class `List<E extends Comparable<E>>` of Chapter 12, our class will organize objects that can be ordered. The class will use the natural ordering provided by the `compareTo` method of the class of objects it organizes or use a different ordering controlled by a `Comparator` object that is provided to it at the time the list is constructed. To keep it as simple as possible, we will not allow duplicate items to be in the list.

An Example of an Ordered List

Let us trace through an example of an ordered list where the data items are simply `String` objects that we wish to maintain in standard alphabetical order. We will insert the following `String` objects: `"meadow"`, `"lake"`, `"zoo"`, and `"tree"`. Note that building an ordered list is a type of sorting, sometimes known as "sorting on the fly". We will assume that each object's data is unique.

When we start building the list, it is empty, and the data item `"meadow"` becomes the first and only item in the list:

We then add the second item, "lake". How do we determine where it goes in the list? In general, we must compare the new item to the existing items in the list. Since lists are linear in nature, we begin by comparing the new item to the first item on the list. In relation to an item on the list, the new item can belong

- directly before the current item (which means that we know exactly where the new item goes)

  or

- somewhere after the current item (which means that we need to continue our comparison process on the remainder of the list)

Comparing "lake" to "meadow", we determine that "lake" comes before "meadow" in alphabetical order. Hence, we have located its place in the list. "lake" becomes the first item in the list, and "meadow" becomes the second item in the list:

```
"lake"
"meadow"
```

We now add "zoo" to the list. Comparing "zoo" to "lake", we see that "zoo" must come somewhere after "lake". We do not know exactly where "zoo" should come after "lake", and hence, we must perform more comparisons. The comparisons we perform will, however, be identical in form to the comparison we just made to "lake". Comparing "zoo" to "meadow", we see that "zoo" must come somewhere after "meadow". We now find ourselves at the end of the list. This case is very similar to the case of an empty list. We simply add the item here.

Our list is now:

```
"lake"
"meadow"
"zoo"
```

We now add "tree" to the list. Comparing "tree" to "lake", we see that "tree" must come somewhere after "lake". Comparing "tree" to "meadow", we see that "tree" must come somewhere after "meadow". Comparing "tree" to "zoo", we see that "tree" must come directly before "zoo". Our list is now:

```
"lake"
"meadow"
"tree"
"zoo"
```

Note that we have located the correct place for the new item in two ways above. Either we have performed a comparison and found that the new item comes before a current list item or we have moved to a point where there are no more list items to be compared with the item being inserted.

Exercise 15.1: In the traces below, note each comparison that is made and the result of that comparison. Your traces should be similar to the discussion above.

a) Trace through the process that would insert the String "ocean" in the list above.

b) Trace through the process that would insert the String "forest" in the list above.

c) Trace through the process that would insert the String "zoology" in the list above.

d) Trace through the process that would insert the String "aardvark" in the list above.

Exercise 15.2: Consider the List<E extends Comparable<E>> class developed in Chapter 12 that implements a list using an array of generic type objects whose class implements the Comparable interface or uses a Comparator supplied at the time the list is constructed. Write a method named addInOrder for the List<E> class. addInOrder has a single parameter of class E. addInOrder accomplishes inserting the parameter into the list in a position appropriate to the ordering scheme. A partial listing of the List<E> class is given below, including the method myCompare, that selects between use of the Comparator instance variable if it is not null and the natural ordering of the objects in the list if the Comparator instance variable is null.

```
import java.util.*;// for the Comparator<E> interface

public class List <E extends Comparable<E>> {

      private E myArray[]; // array of Objects
      private int myCount;// current count of Objects in List
      private Comparator<E> myComparator;// comparator for
                                         // alternate ordering scheme

      private static final int DEFAULT_SIZE = 100;
      public List () {
            myArray = (E []) new Comparable[DEFAULT_SIZE];
            myCount = 0;
            myComparator = null;
      }//0 parameter constructor
      public List (int theSize) {
            myArray = (E []) new Comparable[theSize];
            myCount = 0;
            myComparator = null;
      }//1 int parameter constructor
      public List (Comparator<E> theComparator) {
            myArray = (E []) new Comparable[DEFAULT_SIZE];
            myCount = 0;
            myComparator = theComparator;
```

(continues)

Exercise 15.2, continued

```
        }//1 Comparator parameter constructor
        public List (int theSize, Comparator<E> theComparator) {
              myArray = (E []) new Comparable[theSize];
              myCount = 0;
              myComparator = theComparator;
        }//2 parameter constructor
        private int myCompare (E firstE, E secondE){
              if(myComparator == null) {
                    return firstE.compareTo(secondE);
              }//no comparator
              return myComparator.compare(firstE, secondE);
        }//myCompare

        public boolean add(E addMe) {
              if (!isFull()){
                    myArray[myCount++] = addMe;
                    return true;
              }//add was successful
              System.out.println("Unable to add to full array");
              return false;
        }//add
   }// List<E extends Comparable<E>>
```

An OrderedLinkedList Class

Now let us consider implementing our ordered list with a linked list. We will update the `Node` class developed in Chapter 14 with a 2 parameter constructor. The second parameter will be used to assign the link field appropriately. The updated `Node` A skeleton for that class is displayed below.

```
public class Node<E> {
     private E myData;//refers to the data for this node
     private Node<E> myLink;//refers to the next node in the list
     public Node (E theData) {
          myData = theData;
          myLink = null;
     } // 1 parameter constructor
     public Node (E theData, Node<E> theLink) {
          myData = theData;
          myLink = theLink;
     } // 2 parameter constructor
     public void setLink(Node<E> theLink) {
          myLink = theLink;
     }//setLink
```

```
        public void setData(E theData) {
             myData = theData;
        }//getData

        public Node<E> getLink(){
             return myLink;
        }//getLink

        public E getData() {
             return myData;
        }//getData

    } //Node class
```

Our class named `WCSOrderedLinkedList` will have a reference to the `Node` at the head of the list similar to the situation in our `WCSLinkedList` class in Chapter 14. Hence, our `WCSOrderedLinkedList` class will have the following code skeleton.

```
import java.util.*;//for Comparator<E> interface
public class WCSOrderedLinkedList<E extends Comparable<E>> {
    private Node<E> myHead;
    private Comparator<E> myComparator;// comparator for alternate ordering scheme
    public WCSOrderedLinkedList () {
         myHead = null;
         myComparator = null;
    } // 0 parameter constructor
    public WCSOrderedLinkedList (Comparator<E> theComparator) {
         myHead = null;
         myComparator = theComparator;
    } // 1 parameter constructor
    private int myCompare (E firstE, E secondE){
         if(myComparator == null) {
              return firstE.compareTo(secondE);
         }//no comparator
         return myComparator.compare(firstE, secondE);
    }//myCompare
■■■ //more to come
}  // WCSOrderedLinkedList class
```

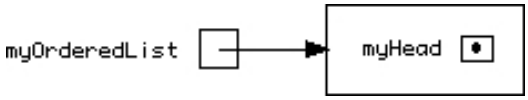
Developing an Algorithm for an Add Method

We will now trace through the formation of an ordered linked list to hold the list of `String` objects we discussed above in order to motivate the Java code for an `add` method for class `WCSOrderedLinkedList`. At this point we will use the natural ordering of `String`s and make our pictures simpler by omitting the `myComparator` instance variable that will be `null` throughout the discussion.

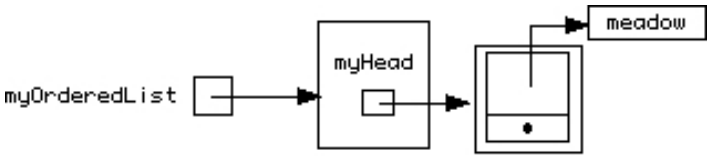
Consider the following Java code:

```
WCSOrderedLinkedList <String> myOrderedList = new WCSOrderedLinkedList < >();
myOrderedList.add("meadow");
myOrderedList.add("lake");
myOrderedList.add("zoo");
myOrderedList.add("tree");
```

After the first line of code is executed, we can draw a sketch of the situation in memory showing an empty `myOrderedList` as follows:

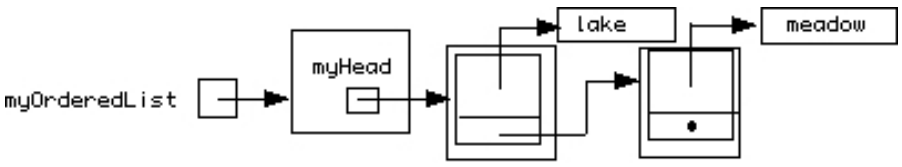


Executing the second line of code, `myOrderedList.add("meadow");`, we see that `myHead` is `null`, indicating an empty list. Hence, we must construct a new `Node` with a reference to the `String` object `"meadow"` in the data field and `null` in the link field, and make `myHead` refer to that newly constructed `Node`. The resulting situation in memory is shown below.



Note that the value of the newly constructed `Node`'s link reference is `null` which is the value of `myHead` before it was assigned to refer to the new `Node`. This observation will prove useful when we write the Java code.

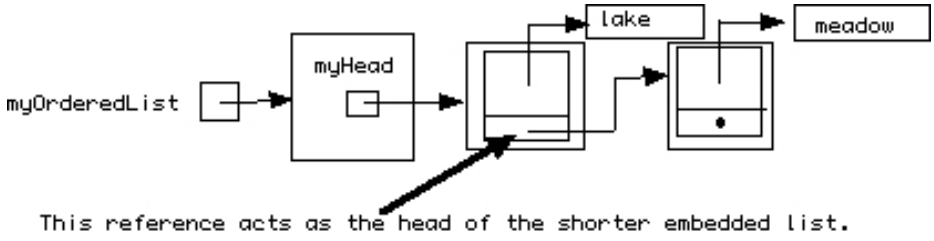
Executing the third line of code, `myOrderedList.add("lake");`, we see that `myHead` is not `null`. Hence, we must compare the `String` object `"lake"` being inserted to the `String` object `"meadow"` at the head of the list. We see that the `String` object `"lake"` should come before the `String` object `"meadow"` at the head of the list. Hence, we construct a new `Node` with the data field assigned the reference to the `String` object `"lake"` and link field assigned the value of `myHead`, and make `myHead` refer to the newly constructed `Node`. The resulting situation in memory is shown below.



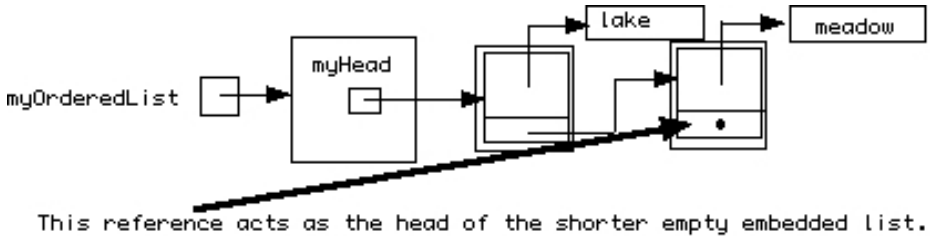
Again the newly constructed `Node` has a link reference that has been assigned the value of `myHead` before `myHead` was assigned to refer to the new `Node`. Note also that `myHead` was assigned an appropriate value as the last step needed in both of the two insertion operations performed so far.

Executing the fourth line of code, `myOrderedList.add("zoo");`, we see that `myHead` is not `null`. Hence, we must compare the `String` object `"zoo"` being inserted to the `String` object `"lake"` at the head of the list. We see that the `String` object `"zoo"` should come somewhere after the `String` object `"lake"`. The key point to notice is that there is another ordered list that is embedded in our current list. That shorter list consists of all

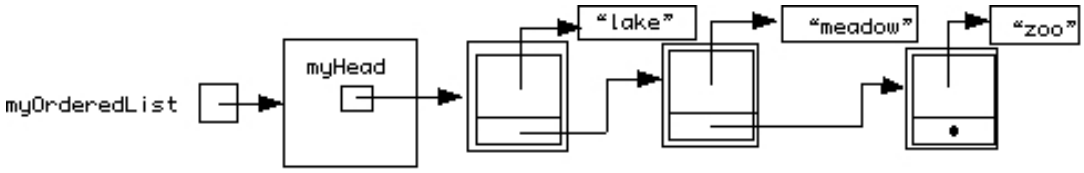
Nodes except the Node at the head of the current list. The link reference of the Node containing a reference to the String object "lake" is a reference to that shorter list that acts as myHead acts for the entire list. See the diagram below.



It is now our job to determine where the String object "zoo" belongs with respect to this shorter list. The String object at the head of the shorter embedded list is "meadow". We compare the String object "zoo" to the String object "meadow" and determine that the String object "zoo" should come somewhere after the String object "meadow." Once again, we consider a second shorter embedded list that consists of all Nodes of the first embedded list except the Node containing the String object "meadow". That shorter embedded list is empty. Notice that the null reference in the Node containing the reference to the String object "meadow" is acting as a reference to an empty list.



Hence, we act as we did when myHead was null above, and we were inserting the String object "meadow" into an empty list. We construct a new Node with data field assigned the reference to the String object "zoo" and link field assigned the value of the head of the current embedded list (null). We must reassign the reference acting as the head of the previously empty embedded list so that it points to the new Node. The resulting situation in memory is shown below.



Note that we used the value of the link field in the "head" Node (in this case the Node containing the reference to "meadow") to initialize the link field of the newly constructed Node.

We can now think of our ordered list as having three embedded shorter ordered lists. They are the ordered list

"meadow"
"zoo"

the ordered list

"zoo"

and the empty list.

Now we execute the fifth line of code, `myOrderedList.add("tree");`. We see that `myHead` is not `null`. Hence, we must compare the `String` object `"tree"` being inserted to the `String` object `"lake"` at the head of the list. We see that the `String` object `"tree"` should come somewhere after the `String` object `"lake"`. In fact, we can say that we should begin the insertion process again with the first shorter embedded list. The link reference in the `Node` containing the reference to "`lake`" acts as the head of this shorter list that consists of the two `Node`s containing the references to "`meadow`" and "`zoo`".

We compare the `String` object `"tree"` to the `String` object `"meadow"` at the head of this first shorter list. We see that the `String` object `"tree"` should come somewhere after the `String` object `"meadow"`. In fact, we can say that we should begin the insertion process again with the second shorter embedded list. The link reference in the `Node` containing the reference to `"meadow"` acts as the head of this second shorter list that consists of the one `Node` containing the reference to `"zoo"`.

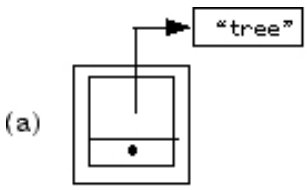
We compare the `String` object `"tree"` to the `String` object `"zoo"` at the head of this second shorter list. We see that the `String` object `"tree"` should come before `"zoo"`. This means that the `String` object `"tree"` should be inserted at the head of the second shorter embedded list. We know the steps to insert a `Node` at the head of a list. We need to

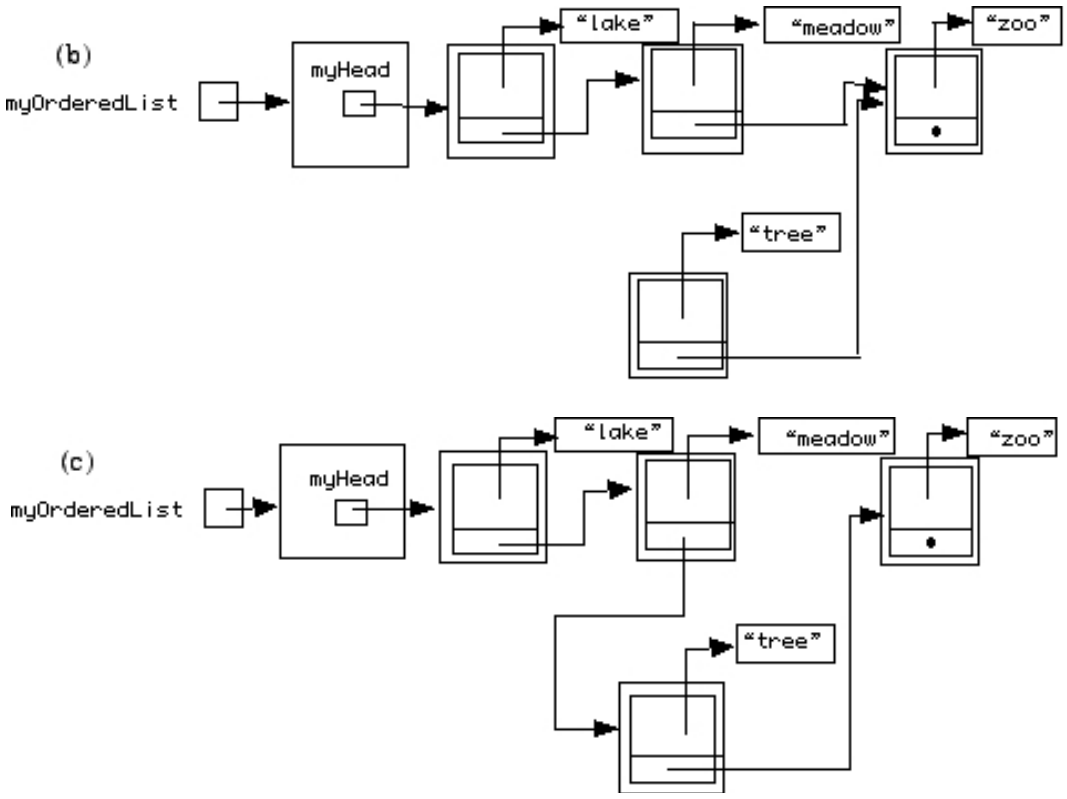
a) construct a new `Node` with data field assigned the reference to the `String` object `"tree"`,

b) make the link reference of the newly constructed `Node` refer to the same thing as the current head of the list refers (in this case the `Node` containing the reference to `"zoo"`) ,

c) and reassign the reference to the `Node` at the head of the list (in this case the link reference in the `Node` containing the reference to `"meadow"`) so that it refers to the newly constructed `Node`. Note that we will need to use the `setLink` method to change the value of this reference if it is the link field of an already existing `Node`.

We give a complete illustration of these three steps in the diagrams below.





Exercise 15.3: Redraw the last picture so that the ordered list appears to flow from left to right. In other words, draw all of the Nodes on one line.

The Java Code for the Add Algorithm

The important feature of the trace above is that we began the insertion process over and over on smaller embedded sub-lists of the initial list until we found the appropriate position for the new Node. It is important to note the conditions under which we found the correct place for the new item, and hence, did not want to begin the insertion process again with a smaller embedded list. Those conditions can be described as follows if we denote the reference to the current smaller embedded list as currentHead.

- The currentHead is null. This means that either the entire list is empty or we have worked our way down to an embedded sub-list that is empty. Hence, the appropriate position for the new Node is at the end of the list.
- The item we are inserting is before the item referenced by currentHead with respect to the agreed upon ordering scheme. This means that the new Node's appropriate position is as the new head of the current list.

Exercise 15.4: In the situation described above where the currentHead is null, explain why this means that the appropriate position for the new Node is at the end of the list.

Now consider the actions that need to be taken when either of the above described conditions occur. We have described the actions several times in the trace above. The actions are:

1) Construct a new Node with the data field assigned the reference to the item to be inserted.
2) Assign the link reference of the newly constructed Node the value of currentHead.
3) Reassign the reference to the list or the embedded sub-list so that it refers to the new Node.

Exercise 15.5: It is important that the second and third steps above be performed in the order presented. What happens if the third step is executed before the second step? Draw a picture to illustrate your answer.

Exercise 15.6: Explain why the three steps above work in both the situation where currentHead is null and the situation where the item we are inserting is before the item in the Node referenced by currentHead.

The *add* method in class WCSOrderedLinkedList will have access to the instance variable myHead. myHead is either null or refers to the first Node in the list. It should be clear from the discussion above that the value of myHead may change during the insertion process.

Exercise 15.7: Describe the situations where the value of myHead changes during insertion of a new item in the list.

The code that invokes the *add* method outside the class WCSOrderedLinkedList will not have access to the private instance variable myHead. When the *add* method is invoked, a reference to the E to be inserted should be passed to the method as a parameter. Hence, we begin with the first line of the public *add* method as follows:

```
public void add (E toInsert) {
```

The fun part of writing the code for the *add* method is to make the code start over and over with smaller embedded sub-lists until the appropriate position for the new Node is found. Previously, we have repeated a process through some type of looping mechanism. This situation is more easily coded using a technique called *recursion*. A method is recursive if it invokes itself. This self-invocation should always modify the arguments to move closer to a solution. In our case the recursive method will have a parameter named currentHead. Each recursive invocation will be sent the "next" currentHead (the reference to the next smaller sub-list).

Normally, recursion is accomplished in Java by writing a pair of methods, one **public** method as described above and one **private** method that accomplishes the recursive processing of the sub-lists. The **public** *add* method is very short because all it does is invoke the **private** *add* method starting at the head of the entire list and assign the reference to the resulting list to **myHead**. The code for the **public** *add* method is as follows:

```
public void add (E toInsert) {
     myHead = add (toInsert, myHead);
}//public add method
```

Note that the **private** *add* method must have two parameters. We certainly can justify sending the reference to the **E** being inserted. We also send a reference to the head of the list so that the **private** method knows where to start checking for the correct position for the new **Node**. We must send **myHead** as a parameter even though it is clearly accessible to the **private** method being written in the same class. The point is that the **private** method will be called recursively on smaller embedded sub-lists if necessary. In those recursive invocations, the reference to the head of the embedded sub-list will be the parameter passed to the **private** method and become the **currentHead** as we discussed in our algorithm.

Because the **public** and **private** methods have different signatures, there will be no confusion about which method is being called. Also note that the return type for the **private** method must be compatible with the type of **myHead** since the value returned by the *add* method will be assigned to **myHead**. The value returned will be the appropriate value for **myHead**. **myHead** will be reassigned its current value if it is not appropriate to change its value.

Compare the code below with the discussion above about when the appropriate position has been found in the list and what the appropriate actions are to be performed. The Java code simply mirrors that discussion for the most part. Note that we use the 2 parameter **Node** constructor to accomplish two algorithm steps in one line of code.

```
private Node<E> add (E toInsert, Node<E> currentHead) {
     if ((currentHead == null) ||
          (myCompare (toInsert, currentHead.getData()) < 0)) {
               Node<E> temp = new Node< > (toInsert, currentHead);
               return temp;
     } //if we have found the correct position
//Otherwise, we need to continue looking for the correct position
//currentHead's link field may be the reference that needs
//to be reassigned to refer to the new Node
     currentHead.setLink(add (toInsert, currentHead.getLink()));
     return currentHead;
}// private add method
```

The **boolean** condition controlling the **if** statement is clearly just a translation into Java syntax of the conditions listed above. The first line in the **if** clause also matches the description of the first two actions that need to be performed. Note, however, that the way that the new value is assigned to the head of the list or sub-list is through a value returned by the method.

Exercise 15.8: Trace the following code to show how a value is assigned to myHead. Be sure to describe the action of each executed statement.

```
WCSOrderedLinkedList<String> myOrderedList =
                            new WCSOrderedLinkedList< >();
myOrderedList.add("meadow");
```

Beyond the `if` statement, we see the recursive invocation of (often called "the recursive call to") the `private` method.

```
currentHead.setLink(add (toInsert, currentHead.getLink()));
```

Note the way in which the method implements the restart of the comparison process with the next smaller embedded sub-list. The second argument sent to the `private` method is the link field of the `Node` at the head of the current sub-list. The recursive invocation of the `private` method appears as the argument for the `setLink` method because the `currentHead`'s link field should refer to the modified list that will exist after this invocation of the `add` method is completed.

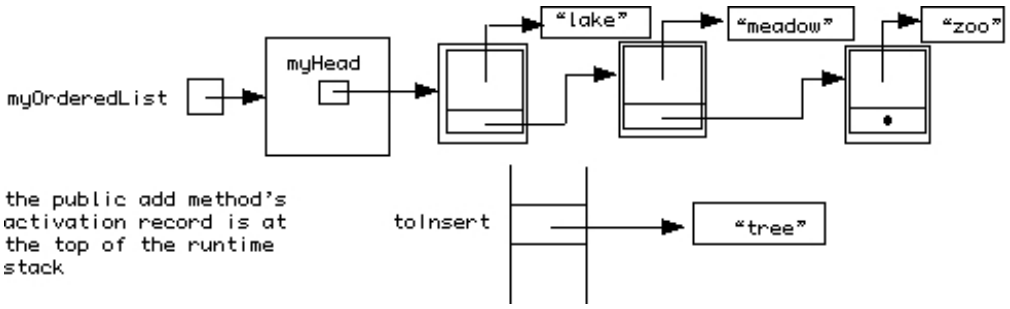
The only tricky part here is that after the invocation of `setLink`, we return the value of `currentHead`. Remember that every call to the `private` method returns to set a link either through a `setLink` method or an assignment statement (for the return from the first call to the `private add` method). Hence, the appropriate value must be returned. Think of the possibilities for that assignment. Either the assignment should be to the reference to the new `Node` as noted in the `if` clause or the assignment should just reassign the current value of the link if that part of the list is to remain unchanged.

Look back at our trace for adding the `String "tree"` to the list that has `"lake"`, `"meadow"`, and `"zoo"` already inserted. The references to the `Node`s containing the references to `"lake"` and `"meadow"` should not be changed. Hence, they should be reassigned their current values. However, the reference to the `Node` containing the reference to `"zoo"` needs to be changed. It needs to be assigned the reference to the new `Node` that contains the reference to `"tree"`. When `currentHead` refers to the `Node` containing the reference to `"meadow"`, it will be appropriate to use `setLink` to change its link field to refer to the new `Node`.

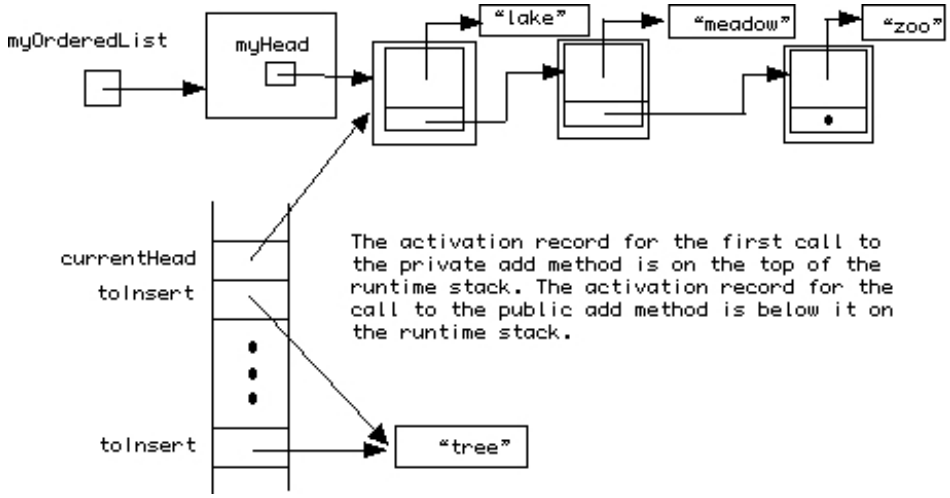
In tracing the code, three values should be returned by the `private add` method. Two of those returned values should have the effect of resetting existing links. The third returned value should reset the link field of the `Node` containing the reference to `"meadow"` so that it refers to the newly constructed `Node` containing the reference to `"tree"`.

The Runtime Stack and Recursion

Let us consider how the runtime stack grows and shrinks as the code executes. When the `public add` method is called, its activation record is pushed onto the runtime stack. Its parameter `toInsert` is referring to the `String` object `"tree"` as shown below.

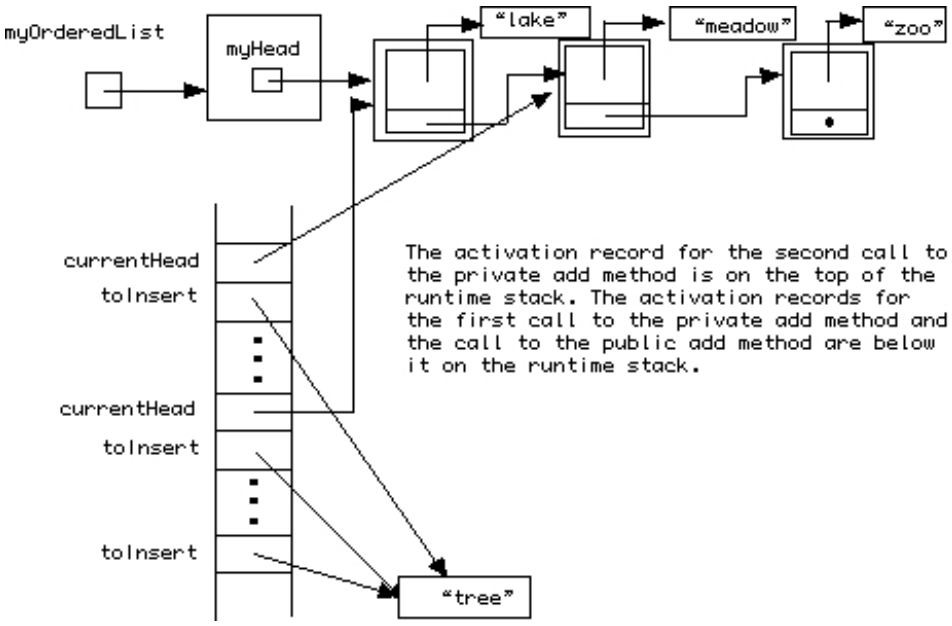


In the first call to the `private add` method from the `public add` method, the `Node` reference sent as an argument is `myHead`. Remember how Java handles parameters. Each parameter is a new copy of the corresponding argument. Hence, when the activation record for the first call to the `private add` method is pushed onto the runtime stack, its parameter `currentHead` is assigned the value of `myHead`. The diagram below illustrates the situation at this point.

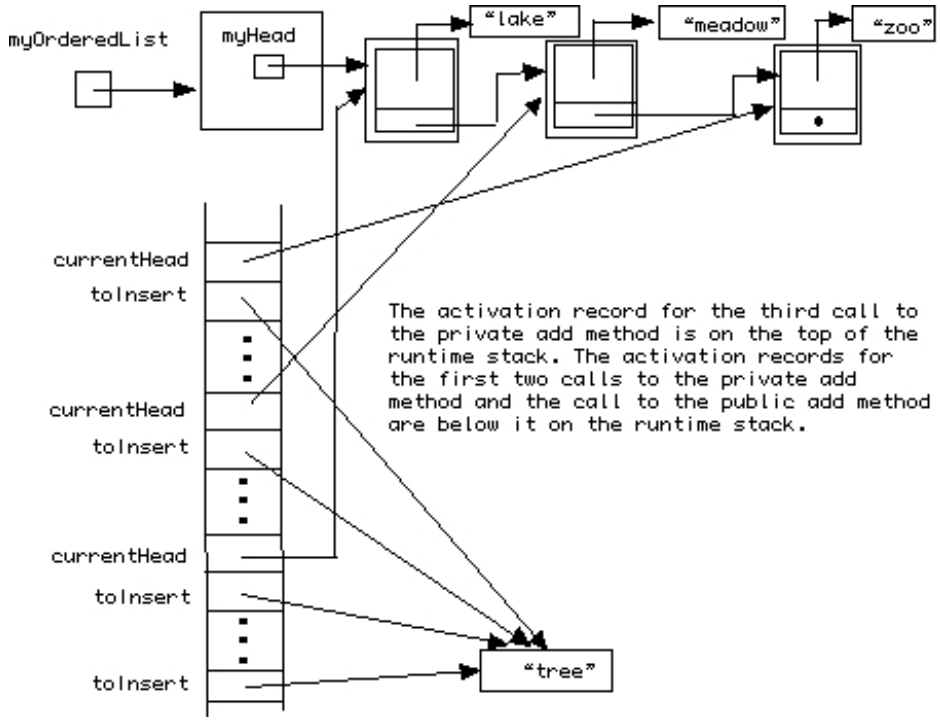


When the `private add` method recurses, that first `currentHead` reference is not destroyed. It exists until that first invocation of the `private add` method completes execution. That execution will not be complete until the second call to the `private add` method returns. That second call to the `private add` method is recursive because it is invoked from within the `private add` method.

In the second call to the `private add` method, the code uses `currentHead.getLink()` for the second argument. This means that the pointer to the `Node` containing the reference to `"meadow"` is the second argument and assigned to the parameter `currentHead` in the next activation record pushed onto the runtime stack for the `private add` method. There are two `currentHead` variables in memory at this point, but there is no confusion because the runtime stack is managing them. The situation in memory is shown below.



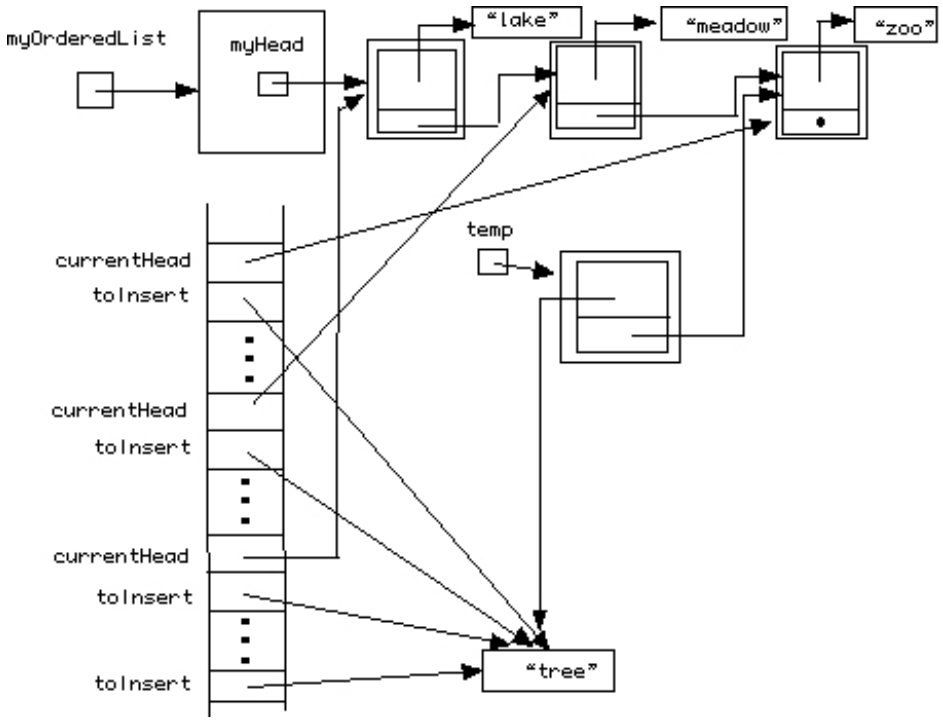
In the third call to the `private add` method, the code uses `currentHead.getLink()` for the second argument. This means that the reference to the `Node` containing the reference to `"zoo"` is the second argument and assigned to the parameter `currentHead` in the next activation record pushed onto the runtime stack for the `private add` method. There are three `currentHead` variables in memory at this point. The situation in memory is shown below.



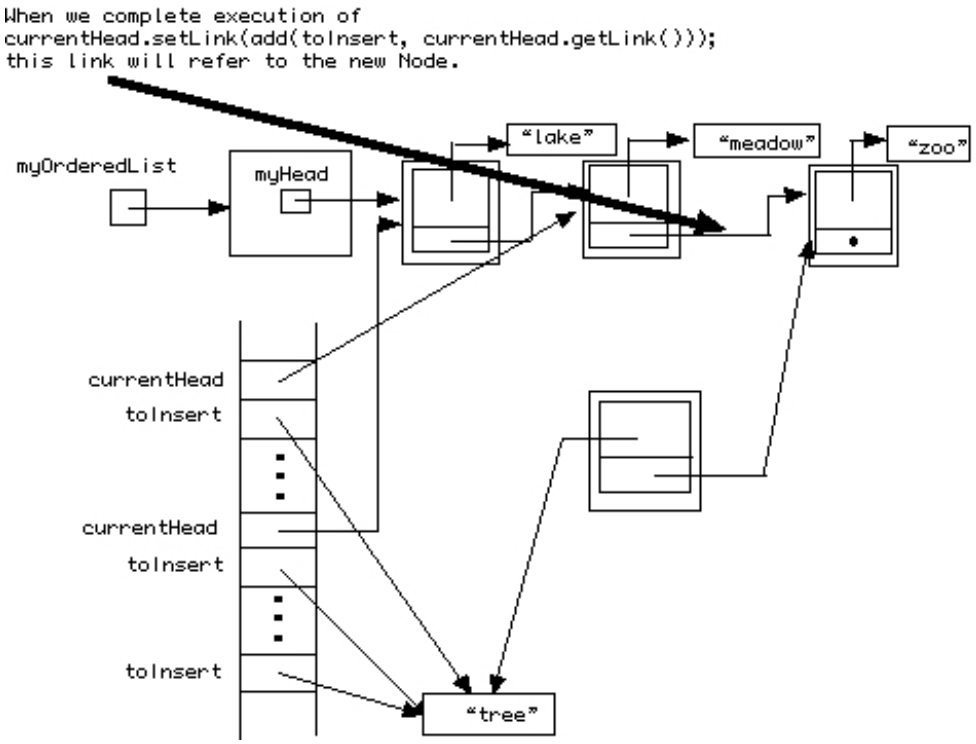
Finally, in this third call to the `private add` method, the condition

```
(myCompare (toInsert, currentHead.getData()) < 0)
```

is `true` so that we will not recurse any more. A `Node` is constructed with data field assigned the reference to the `String` object `"tree"`. That `Node`'s link field is assigned to the value of `currentHead`. The situation in memory is shown below just before the `return` statement.

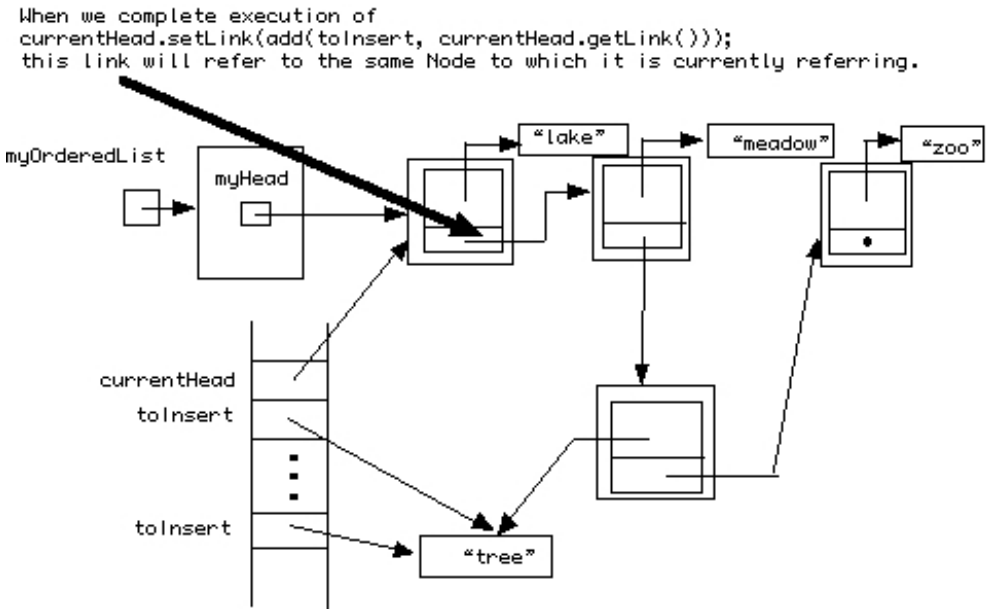


Now, we return the value of `temp` and pop the top activation record off the runtime stack. `temp` will no longer exist, but we must remember that its value is being returned to be used. Note that the value returned is a reference to the new `Node`. We return to the second call to the `private add` method and are ready to complete execution of the invocation of the `setLink` method using the returned value for its argument. The diagram below shows the situation just before that `setLink` takes place.



After completing the `setLink`, the new `Node` will be appropriately linked into the list.

Then it remains to `return currentHead` to complete the second invocation of the `private add` method. Its activation record will be popped off the runtime stack. The value returned at this point is a reference to the `Node` containing the reference to `"meadow"`. That value will be the argument to the `setLink` method of the first call to the `private` method and will reassign the link field in the `Node` containing the reference to `"lake"` so that it still points to the `Node` containing `"meadow"`. The situation in memory just before that `setLink` is executed is shown below.



After completing the `setLink`, the link field of the `Node` containing the reference to `"lake"` is still referring to the `Node` containing the reference to `"meadow"`.

Then it remains to `return currentHead` to complete the first invocation of the `private add` method. Its activation record will be popped off the runtime stack leaving only the activation record of the `public add` method. The value returned at this point is a reference to the `Node` containing the reference to `"lake"`. Execution returns to complete the assignment statement in the `public add` method. The returned value will be assigned to `myHead`. Hence, `myHead` is reassigned so that it continues to refer to the `Node` containing the reference to `"lake"`.

Exercise 15.9: Trace the recursion that results from the following line of code being executed on the ordered list that contains `"lake"`, `"meadow"`, `"tree"`, and `"zoo"`.

```
myOrderedList.add("turkey");
```

Draw pictures showing the situation in memory (including the runtime stack and the linked list) at each of the following points in the execution:

- whenever an activation record is pushed onto the runtime stack
- just before completing a `setLink` invocation.

Exercise 15.10: We have ignored the possibility of insertion of duplicate items into the list. Trace the recursion that results from the following line of code being executed on the ordered list that contains `"lake"`, `"meadow"`, and `"zoo"`.

```
myOrderedList.add("meadow");
```

Draw pictures as described in Exercise 15.9.

Building a WCSOrderedLinkedList without Duplicates

We would like to modify our `public` and `private add` methods so that duplicate items are not added to the list. Let us first consider the `private add` method. Remember that as the recursion proceeds, we test to determine whether a new `Node` needs to be constructed and a reference to that newly constructed `Node` returned, or if the recursion should proceed to the next `Node` and the `currentHead` should be returned. The first question to answer is "What should be returned if a duplicate item is found in the list?" It should be clear that we simply want to leave the list as it is. Hence, there is no construction of a new `Node`, no recursive call to continue searching for the correct position in the list, and the `currentHead` should be returned. In other words, if a duplicate item is found to be already in the list, we just return the `currentHead`.

The condition that determines that a new `Node` should be constructed is tested first in our `private add` method. That condition is:

```
((currentHead == null) || (myCompare (toInsert, currentHead.getData()) < 0))
```

Remember that this condition tests both for getting to the end of the list or finding that the item being added is before an item already in the list. However, notice that if `toInsert` and `currentHead.myData` are in fact referring to equal items, this condition is `false`, and the recursion goes on to the next `Node` in the list. Thus, if we want to avoid having more than one `Node` referring to equal items, we need to separate the tests and make sure that we check for duplicate items before we test using the `myCompare` method.

We should always check for the end of the list first to make sure that our code avoids references through `null` pointers. In summary, our code should check first for the end of the list, should check second for a duplicate item, and should finally check using the `myCompare` method to see if the correct position in the list has been found. If all three of these conditions fail, it is then time to recurse down the list.

We have assembled a new version of the `private add` method below that conforms to the discussion in the last few paragraphs. The `Node<E>` reference `temp` was not really needed and has been avoided in our new version. Note that we once again use Java 1.7's diamond operator in the `return` statements. The compiler can infer the type parameter `E` for the construction of the `new Node` from the return type `Node<E>` of the method.

```
private Node<E> add (E toInsert, Node<E> currentHead) {
     if (currentHead == null) {
          return new Node< > (toInsert);
     } // We got to the end of the list.
     if (toInsert.equals(currentHead.getData())) {
          return currentHead;
     } // We found a duplicate item.
     if (myCompare (toInsert, currentHead.getData()) < 0) {
          return new Node< > (toInsert, currentHead);
     } // We found the correct position.
     //Otherwise, we need to continue looking for the correct position
     //currentHead's link field may be the reference that needs
     //to be reassigned to refer to the new Node
     currentHead.setLink(add (toInsert, currentHead.getLink()));
     return currentHead;
}// private add method
```

Another consideration with respect to duplicate items is whether the code somehow indicates that a new item was added or not. We would like our `public add` method to have return type `boolean` and return `true` if the list's composition was changed by the actions of the `add` methods or return `false` if the list was unchanged by the actions of the `add` methods. The easiest way to determine if the list has changed or not is to have a `boolean` instance variable in the `WCSOrderedLinkedList` class. Let us name that variable `addSuccessful`. `addSuccessful` is assigned `false` by the `public add` method before it begins the recursion. At any point where the `private add` method constructs a new `Node` that will change the composition of the list, `addSuccessful` should be assigned the value `true`. Finally, the `public add` method should return the value of `addSuccessful`. We have assembled the improved versions of the `public` and `private add` methods below.

```
// With the instance variables for the class
private boolean addSuccessful;// used to indicate that the add methods
                              // changed the composition of the list or not

public boolean add (E toInsert) {
     addSuccessful = false;
     myHead = add (toInsert, myHead);
     return addSuccessful;
}//public add method

private Node<E> add (E toInsert, Node<E> currentHead) {
     if (currentHead == null) {
          addSuccessful = true;
          return new Node< > (toInsert);
     } // We got to the end of the list.
     if (toInsert.equals(currentHead.getData())) {
          return currentHead;
     } // We found a duplicate item.
     if (myCompare (toInsert, currentHead.getData()) < 0) {
          addSuccessful = true;
          return new Node< > (toInsert, currentHead);
     } // We found the correct position.
     //we need to continue looking for the correct position
     //currentHead's link field may be the pointer that needs
     //to be reassigned to point to the new Node
     currentHead.setLink(add (toInsert, currentHead.getLink()));
     return currentHead;
}// private add method
```

Recall from our trace of this recursive code that there are several references to the object `toInsert` on the runtime stack. There is such a reference in each activation record for the `private add` method. We can avoid that waste of space on the runtime stack if we can make the reference available to the `private add` method in some way other than as a parameter. One way to do that is to have another `private instance` variable of class `E` that will hold the reference. Let us name that `private` instance variable `objectBeingAdded`. `objectBeingAdded` will be assigned in the `public add` method. It will be assigned the reference passed as a parameter to the `public add` method. Then the `private add` method will simply refer to the item being added as `objectBeingAdded`. Taking these improvements into consideration, our code becomes:

```
// With the instance variables for the class
private boolean addSuccessful;// used to indicate that the add methods
                              // changed the composition of the list or not
private E objectBeingAdded; // holds a reference to object
                                    // being added for the recursion

public boolean add (E toInsert) {
     addSuccessful = false;
```

```
        objectBeingAdded = toInsert;
        myHead = add (myHead);
        return addSuccessful;
    }//public add method

    private Node<E> add (Node<E> currentHead) {
        if (currentHead == null) {
            addSuccessful = true;
            return new Node< > (objectBeingAdded);
        } // We got to the end of the list.
        if (objectBeingAdded.equals(currentHead.getData())) {
            return currentHead;
        } // We found a duplicate item.
        if (myCompare (objectBeingAdded, currentHead.getData()) < 0) {
            addSuccessful = true;
            return new Node< > (objectBeingAdded, currentHead);
        } // We found the correct position.
        //we need to continue looking for the correct position
        //currentHead's link field may be the pointer that needs
        //to be reassigned to point to the new Node
        currentHead.setLink(add (currentHead.getLink()));
        return currentHead;
    }// private add method
```

Searching a WCSOrderedLinkedList

Another typical list operation is searching an ordered linked list for a particular item. Since the list is ordered, the code should be able to detect that an item is not in the list by finding an item in the list that is after the item for which it is searching.

Think about the search process recursively, i.e., in terms of searching smaller and smaller embedded sub-lists. The algorithm should compare the item at the head of the list with the search item. If the current list is empty or the item at the head of the current sub-list is after the search item, return `null` to indicate that the item is not in the list. If the item at the head of the list is the search item, return a reference to the item. If the conditions just stated are all false, it must be that the item at the head of the list is before the search item. In that case, return the result of the recursive call on the next smaller embedded sub-list.

Once again, we use a pair of methods. The `public` method will be passed a reference to the search item. It will invoke the `private` method and return whatever the private method returns to it. The code for the `public search` method uses a private instance variable of class `E` named `objectSearchingFor` and is as follows:

```
    public E search(E findMe) {
        objectSearchingFor = findMe;
        return search(myHead);
    }// public search
```

Note that the code for searching should not change the structure of the list. Hence, there will be no assignment statements or `setLink` invocations involved.

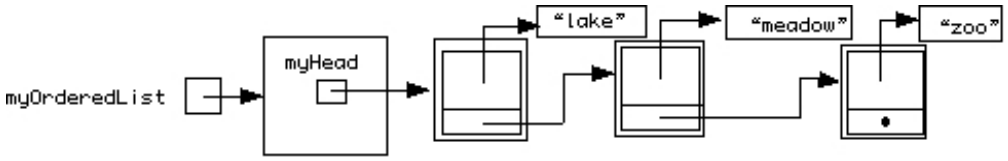
Exercise 15.11: Write the code for the `private search` method using the header below and the algorithm described above.

```
private E search (Node<E> currentHead) {
```

Deleting an Item from a WCSOrderedLinkedList

Let us now consider deleting an item from an ordered linked list. If the item being deleted is at the head of the list, `myHead` will be reassigned. We can use that description in a recursive process by saying that if the item being deleted is at the head of the current sub-list, the reference to that current sub-list will be reassigned. We know from our discussion of the *add* method that reassigning links can be accomplished with an assignment statement or with a call to `setLink` on the appropriate `Node`.

Consider deleting the `Node` containing the reference to `"meadow"` from the list below.



It is the link reference in the `Node` containing the reference to `"lake"` that must be reassigned so that it refers to the `Node` containing the reference to `"zoo"`. Recall that in the code for the *add* methods, we returned the appropriate link reference whether it was a link to remain in place or a link to a new `Node`. When the `Node` containing the reference to `"lake"` is at the head of the current sub-list, it is `currentHead` that needs to be returned in order to keep this `Node` in the list. When the `Node` containing `"meadow"` is at the head of the current sub-list, it is the value of its link field that needs to be returned in order to delete this `Node`.

It may be that the item to be deleted is not found in the list. The `public delete` method should have `boolean` return type. The value `true` should be returned if the composition of the list has been changed by the actions of the `public` and `private delete` methods. The value `false` should be returned by the `public delete` method if the composition of the list is unchanged by the actions of the `delete` methods. We will add another `boolean` instance variable and another instance variable of class `E` to the `WCSrderedLinkedList` class to accomplish this task. We name the `boolean` instance variable `deleteSuccessful`. We name the instance variable of class `E` as `objectToDelete`.

Exercise 15.12: The `public delete` method should be very similar to the `public` *add* method. Write the `public delete` method assuming that the class has a `private boolean` instance variable named `deleteSuccessful` and a `private` instance variable of class `E` named `objectToDelete`.

Exercise 15.13: The `private delete` method should assign the value `true` to `deleteSuccessful` when the item to be deleted is found and the appropriate link returned. Also note that the `private delete` method should not recurse if it encounters a `Node` containing an item that is *after* the item to be deleted. Write the code for the recursive `private delete` method.

Exercise 15.14: Complete the class named `WCSOrderedLinkedList<E extends Comparable<E>>` that implements a linked list of items that is maintained in order.

Your `WCSOrderedLinkedList <E extends Comparable<E>>` class should have at least eight private instance variables. The first instance variable is of class `Node<E>` and is a reference to the head of the list. The second instance variable is of class `Comparator<E>` and is used to maintain the list in an order specified by the `Comparator`. The third instance variable keeps a count of items in the list. The count should be updated every time an add or delete operation is successful. The remaining five instance variables have been described above in discussions of adding, searching, and deleting from the list.

One constructor of class `WCSOrderedLinkedList <E extends Comparable<E>>` will have one parameter that is a reference to a `Comparator<E>` object. The other constructor will have no parameters.

Your class `WCSOrderedLinkedList <E extends Comparable<E>>` should have at least the following instance methods:

```
public boolean add(E addMe)
private Node<E> add(Node<E> currrentHead)
public E search(E findMe)
private E search(Node<E> currrentHead)
public boolean delete(E deleteMe)
private Node<E> delete(Node<E> currentHead)
public boolean isEmpty()
public int getCount()
private int myCompare(E firstItem, E secondItem)
public String saveToFile(String fileName)
private void saveToFile(Node<E> currentHead)
public String loadFromFile(String fileName)
public String toString()
private String toString(Node<E> currentHead)
```

You will need a `Node<E>` class to work in conjunction with your `WCSOrderedLinkedList <E extends Comparable<E>>` class.

Exercise 15.15: Continuation of Exercise 14.22

a) In your Address Database GUI, you have buttons that allow the user to display the `AddressInfo` objects in the usual two possible orders, but with an ordered linked list, the underlying data structure will *not* be sorted as a consequence of the user's having clicked one of these buttons. Instead the data in each data structure will be maintained in order. Hence, you should change the text that appears on those buttons to "Alphabetical Order Display" and "Address Order Display". You should also remove the original "Display" button from the GUI and remove any code pertaining to that button from your `Database` class.

b) Revise your `Database` class so that you declare, construct, and use two `WCSOrderedLinkedList <AddressInfo>` objects, one that can be ordered alphabetically and one that can be ordered by address. When you construct your `WCSOrderedLinkedList <AddressInfo>` that can be ordered by address, you should pass the public reference to the `AddressComp` object to the constructor.

c) When you add a new record to the database, you add it to both `WCSOrderedLinkedList < AddressInfo>` objects.

d) When you delete a record from the database, you delete it from both `WCSOrderedLinkedList < AddressInfo>` objects.

e) When your code saves the contents of the database to a file, save only the items in the alphabetical list.

f) When your code loads items from a file, add each item to both lists.

Exercise 15.16: Continuation of Exercise 14.23

a) In your Birthday Database GUI, you have buttons that allow the user to display the `BirthInfo` objects in the usual two possible orders, but with an ordered linked list, the underlying data structure will *not* be sorted as a consequence of the user's having clicked one of these buttons. Instead the data in each data structure will be maintained in order. Hence, you should change the text that appears on those buttons to "Alphabetical Order Display" and "Chronological Order Display". You should also remove the original "Display" button from the GUI and remove any code pertaining to that button from your `Database` class.

b) Revise your `Database` class so that you declare, construct, and use two `WCSOrderedLinkedList <BirthInfo>` objects, one that can be ordered alphabetically and one that can be ordered chronologically. When you construct your `WCSOrderedLinkedList <BirthInfo>` object that can be ordered chronologically, you should pass the public reference to the `ChronComp` object to the constructor.

c) When you add a new record to the database, you add it to both `WCSOrderedLinkedList <BirthInfo>` objects.

(continues)

Exercise 15.16 continued

d) When you delete a record from the database, you delete it from both `WCSOrderedLinkedList <BirthInfo>` objects.

e) When your code saves the contents of the database to a file, save only the items in the alphabetical list.

f) When your code loads items from a file, add each item to both lists.

Exercise 15.17: Continuation of Exercise 15.15

In this exercise you will separate the code in the Address Database GUI from the choice of type of data structure that manages the data. You will hide the fact that there are two underlying lists.

a) Write a class named `AddressInfoCollection` that relies on two ordered linked lists to organize the `AddressInfo` objects. More specifically:

   i) Your class `AddressInfoCollection` will have two private instance variables that are references to objects of class `WCSOrderedLinkedList <AddressInfo>`. One `WCSOrderedLinkedList <AddressInfo>` will be constructed with zero parameter constructor, and hence be ordered by the natural ordering of `AddressInfo` objects. The second will be constructed with the `AddressComp` object of class `AddressInfo` as an argument to the constructor, and hence, be ordered by address.

   ii) Class `AddressInfoCollection` will have the following methods:

      (1) `public AddressInfo search(AddressInfo)`—This `search` method will return the value returned by the `search` method invoked on the `WCSOrderedLinkedList <AddressInfo>` ordered by the natural ordering of `AddressInfo` objects.

      (2) `public boolean add(AddressInfo)`—This `add` method will invoke the `search` method described above to check for a duplicate record. If a duplicate is found, `false` is returned. If no duplicate is found, this `add` method will return `true` only if the attempts to add the record into both `WCSOrderedLinkedList <AddressInfo>` 's are successful. It will return `false` otherwise.

      (3) `public boolean delete(AddressInfo)`—This `delete` method will invoke the `search` method described above and assign the returned value to a local `AddressInfo` reference for the `found` object. If the returned value is `null`, the `delete` method should return `false`. If the returned value is not `null`, `delete` should return `true` only if the attempts to delete the `found` reference from both `WCSOrderedLinkedList <AddressInfo>`'s are successful. The `delete` method should return `false` otherwise.

      (4) `public String toStringAlphabetical()`—This method returns the value returned by invoking `toString()` on the `WCSOrderedLinkedList <AddressInfo>` that is ordered by the natural ordering of `AddressInfo` objects.

      (5) `public String toStringByAddress()`—This method returns the value returned by invoking `toString()` on the `WCSOrderedLinkedList <AddressInfo>` that is ordered by the `AddressComp` object of class `AddressInfo.`

(continues)

Exercise 15.17, continued

(6) `public String saveToFile(String)`—This method returns the `String` returned by invoking `saveToFile` on the `WCSOrderedLinkedList <AddressInfo>` that is ordered by the natural ordering of `AddressInfo` objects with argument equal to the `String` parameter that is a file name.

(7) `public String loadFromFile(String)`—This method will be similar to the method `loadFromFile` in the class `List<E extends Comparable<E>>` class from Chapter 13. However, in this case, we do *not* want to invoke `loadFromFile` on each of the two `WCSOrderedLinkedList <AddressInfo>` s. If we did that, we would have two sets of the same data objects in memory that were read from the file. Instead, write `loadFromFile` in the class `AddressInfoCollection` so that each object comes just once from the file and is inserted into both `WCSOrderedLinkedList <AddressInfo>` s using the add method described above.

b) Add your class `AddressInfoCollection` to your Eclipse project.

c) In your `Database` class declare a reference to an object of class `AddressInfoCollection` with name `myCollection`.

d) In the constructor of your `Database` class construct the `AddressInfoCollection` and make `myCollection` refer to it with the following line of code:

```
myCollection = new AddressInfoCollection();
```

e) Remove the declarations and constructions of the two `WCSOrderedLinkedList <AddressInfo>` objects. There will be many errors generated in your `Database` class due to this action. However, those red flags indicate the places where the object `myCollection` should be used and the methods of `AddressInfoCollection` should be invoked. Make the appropriate changes so that your Address Database application works correctly.

Exercise 15.18: Continuation of Exercise 15.16

In this exercise you will separate the code in the Birthday Database GUI from the choice of type of data structure that manages the data. You will hide the fact that there are two underlying lists.

a) Write a class named `BirthInfoCollection` that relies on two ordered linked lists to organize the `BirthInfo` objects. More specifically:

i) Your class `BirthInfoCollection` will have two private instance variables that are references to objects of class `WCSOrderedLinkedList <BirthInfo>`. One `WCSOrderedLinkedList <BirthInfo>` will be constructed with zero parameter constructor, and hence be ordered by the natural ordering of `BirthInfo` objects. The second will be constructed with the `ChronComp` object of class `BirthInfo` as an argument to the constructor, and hence, be ordered chronologically.

ii) Class `BirthInfoCollection` will have the following methods:

(1) `public BirthInfo search(BirthInfo)`—This search method will return the value returned by the search method invoked on the `WCSOrderedLinkedList <BirthInfo>` ordered by the natural ordering of `BirthInfo` objects.

(continues)

Exercise 15.18, continued

(2) `public boolean add(BirthInfo)`—This `add` method will invoke the `search` method described above to check for a duplicate record. If a duplicate is found, `false` is returned. If no duplicate is found, this `add` method will return `true` only if the attempts to add the record into both `WCSOrderedLinkedList <BirthInfo>`'s are successful. It will return `false` otherwise.

(3) `public boolean delete(BirthInfo)`—This `delete` method will invoke the `search` method described above and assign the returned value to a local `BirthInfo` reference for the `found` object. If the returned value is `null`, the `delete` method should return `false`. If the returned value is not `null`, `delete` should return `true` only if the attempts to delete the `found` reference from both `WCSOrderedLinkedList <BirthInfo>`'s are successful. The `delete` method should return `false` otherwise.

(4) `public String toStringAlphabetical()`—This method returns the value returned by invoking `toString()` on the `WCSOrderedLinkedList <BirthInfo>` that is ordered by the natural ordering of `BirthInfo` objects.

(5) `public String toStringChronological()`—This method returns the value returned by invoking `toString()` on the `WCSOrderedLinkedList <BirthInfo>` that is ordered by the `ChronComp` object of class `BirthInfo.`

(6) `public String saveToFile(String)`—This method returns the `String` returned by invoking `saveToFile` on the `WCSOrderedLinkedList <BirthInfo>` that is ordered by the natural ordering of `BirthInfo` objects with argument equal to the `String` parameter that is a file name.

(7) `public String loadFromFile(String)`—This method will be similar to the method `loadFromFile` in the class `List<E extends Comparable<E>>` class from Chapter 13. However, in this case, we do *not* want to invoke `loadFromFile` on each of the two `WCSOrderedLinkedList <BirthInfo>`'s. If we did that, we would have two sets of the same data objects in memory that were read from the file. Instead, write `loadFromFile` in the class `BirthInfoCollection` so that each object comes just once from the file and is inserted into both `WCSOrderedLinkedList <BirthInfo>`'s using the add method described above.

b) Add your class `BirthInfoCollection` to your Eclipse project.

c) In your `Database` class declare a reference to an object of class `BirthInfoCollection` with name `myCollection`.

d) In the constructor of your `Database` class construct the `BirthInfoCollection` and make `myCollection` refer to it with the following line of code:

```
myCollection = new BirthInfoCollection();
```

e) Remove the declarations and constructions of the two `WCSOrderedLinkedList <BirthInfo>` objects. There will be many errors generated in your `Database` class due to this action. However, those red flags indicate the places where the object `myCollection` should be used and the methods of `BirthInfoCollection` should be invoked. Make the appropriate changes so that your Birthday Database application works correctly.

Lab: Linked List Lab 2 • Chapter 15

The objective of this lab is:

- to get practice writing and testing methods that work with linked list data structures

This lab uses `.java` files in a folder entitled `for_linked_list_lab2`

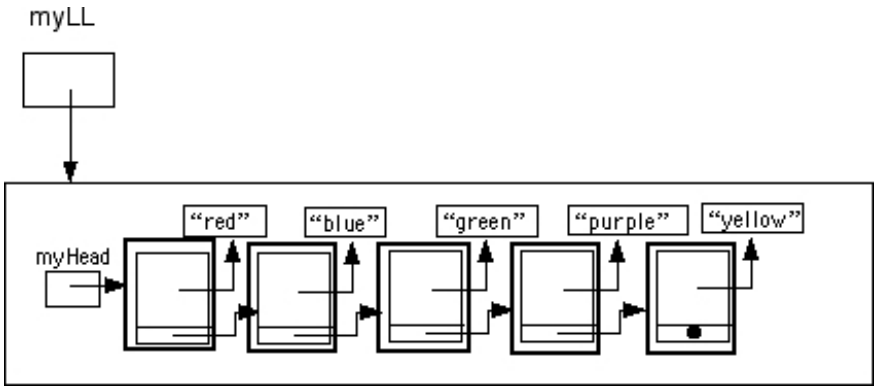
1) Get a copy of the folder named `for_linked_list_lab2` onto the **Desktop**.
2) Launch Eclipse.
3) Create a new Java project named with `LinkedListLab2` **and** `your name`.
4) Drag the folder `for_linked_list_lab2` from the **Desktop** onto the image of the your `LinkedListLab2` folder in the `Package Explorer` view.
5) Double click on your `LinkedListLab2` folder in the `Package Explorer` view to open the folder.
6) Double click on the `for_linked_list_lab2` package in the `Package Explorer` view to open the folder. You should see that the files `TestWCSLab2LinkedList.java`, `WCSLab2LinkedList.java` and `Node.java` in the `for_linked_list_lab2` package.

Write each of the seven methods described below. You will be adding those methods to the `WCSLab2LinkedList` class and testing those methods by writing code in the `main` method of class `TestWCSLab2LinkedList`.

When you are asked to print, use `System.out.println.`

- Include a brief comment at the beginning of each method giving the step number and describing what the method does.
- Make sure your output statements make clear which method is being invoked and what, if any, arguments it was passed.

To understand the problems given below, the following linked list is given as a *sample*:



7) Write a method named `getItemCount` that returns the number of items in the linked list. The method should *NOT* have any parameters.

8) Write a method named `numberedToString` that returns a `String` that displays each element in the list with its position. The method should *NOT* have any parameters. For the list above `numberedToString` should return a `String` beginning with `"1. red\n 2.blue\n…"`

9) Consider the linked list above. The `Node` referring to "`red`" could be considered the first `Node` in the list. The `Node` referring to "`blue`" could be considered the second `Node` in the list, and so on. Write a method named `dataAtPos` that has a single `int` parameter representing a position in the list. The method should return the data at that position or `null` if the position is not in the range 1 through the item count for the linked list. For example, your Java code should output `"The data at position 4 is purple."` for the list shown above.

10) Write a method named `positionFirstOccurrence` that has a single `String` parameter representing possible data in the list. The method should return the first position at which the data occurs, or `0` if the data does not occur in the list. The following Java code would output `"green is at position 3."`

```
int position = myLL.positionOf("green");
System.out.println("green is " + (position == 0 ? "not in the list."
                                  :( "at position " + position)));
```

11) Write a method named `rotate` that moves the last `Node` in the linked list so that it becomes the first `Node` in the linked list. The method has no parameters and returns no value. Before invoking the method you should print the existing list. After invoking the method, you should print the newly modified list.

12) Write a method named `countMatches` that has a single parameter and returns an `int` representing the number of times the parameter appears in the linked list.

13) Write a method named `reverse`, that modifies the list so that the data is reversed. The method has no parameters and returns no value. Before invoking the method you should print the existing list. After invoking the method, you should print the newly modified list.

14) Submit hard copies of the classes `WCSLab2LinkedList` and `TestWCSLab2LinkedList`. Also, submit a hard copy of the final output in Eclipse's Console view.

16

Binary Trees

INTRODUCTION

The structure that we call a tree should be familiar to you from various contexts. For instance, everyone has a "family tree". Biologists classify plants and animals according to a scheme that can be represented as a tree. Most companies have "organizational charts" that display the hierarchy of responsibility in the company. Usually, such a chart takes the form of a tree. In fact, there are many circumstances where the relationships between objects is well described by some sort of tree. Hence, it should not be surprising that constructing tree-shaped dynamic data structures turns out to be useful.

Some Definitions

Informally, what we mean by a tree is a structure with form something like any of the shapes in Figure 16.1:

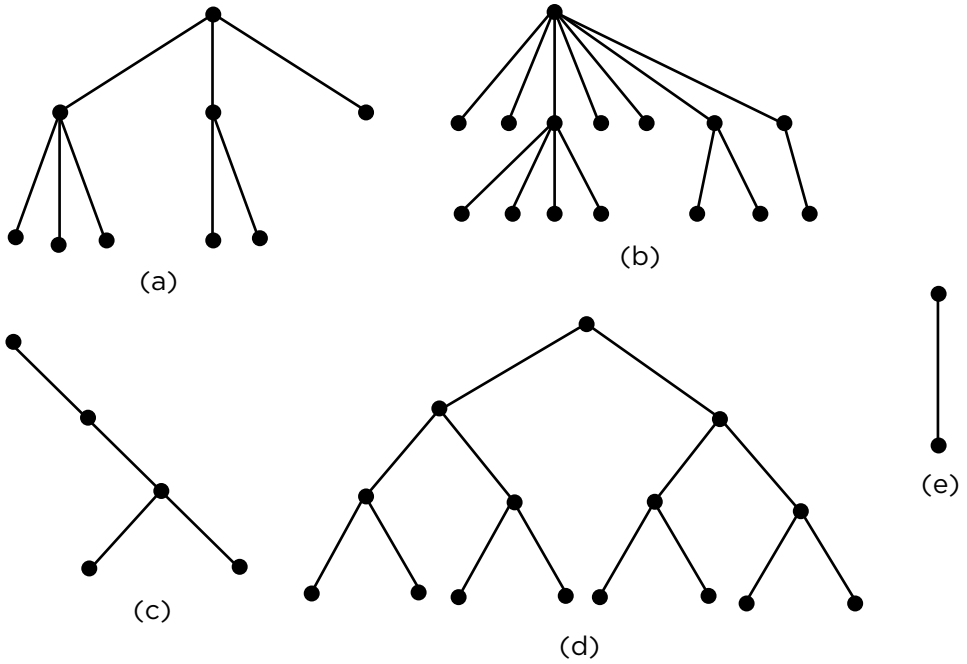


Figure 16.1 Various Shapes for Trees

There is a suggestive, but possibly confusing, vocabulary for describing the parts of a tree structure. Think of the lines as emanating from upper to lower (left or right) dots. Then the top (as we have drawn it) of a tree, from which zero or more lines emanate, is referred to as the *root*. All the lines are referred to as *branches*. Thus, as other writers have remarked, computer scientists draw their trees upside down. The various places from which branches emanate are referred to as *nodes*. The root counts as a node. The ends of branches from which no new branches emanate are also referred to as nodes, but in addition they have the special name *leaves*. If a branch goes from node A to node B, then node A is node B's *parent*, while node B is node A's *child*. A node can have any number of children, but every node except the root must have exactly one parent. Thus the shapes in Figure 16.2 are NOT trees. (In 16.2a there is a node with two parents, and in 16.2b there is a non-root node with no parent.)

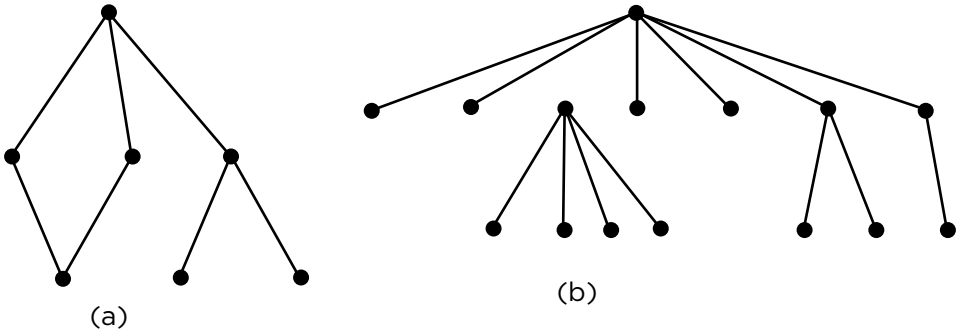


Figure 16.2 Shapes That Are Not Trees

The depth of a tree is the number of nodes in any of the longest paths from the root to a leaf. Both the root and the leaf count in the depth.

Exercise 16.1: Give the depth of each of the trees in Figure 16.1.

Binary Trees

We are only going to discuss the special kind of tree known as the binary tree in this text. Even our discussion of the binary tree will be limited. It turns out that tree structures, and binary trees in particular, are so useful in computer science that the theory regarding their manipulation has been extensively studied and developed. This theory is rich and complex. Much of it is typically covered in courses devoted to the study of Data Structures. We will content ourselves with some definitions, a few examples, and a hint of the deeper problems.

Informally, a binary tree is a tree in which every node has at most two children. The formal definition of a binary tree is a recursive one. This is no accident. The recursive nature of binary trees is reflected in the recursive nature of many of the algorithms for manipulating them. A binary tree is a finite data structure which either is empty, or consists of a node with references to two binary trees. The binary trees attached to a node are its sub-trees. There is a left sub-tree and a right sub-tree at each node; either or both of the sub-trees may be empty.

Exercise 16.2: Which if the trees shown in Figure 16.1 are binary trees, according to the above definition?

In a sense, the tree structure is a generalization of the list: we can think of a linked list as a "degenerate" binary tree in which all the left sub-trees at each `TreeNode` are empty as in Figure 16.3 below.

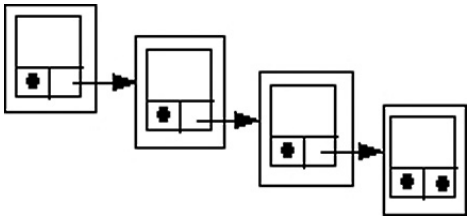


Figure 16.3 Linked List as Degenerate Binary Tree

The distinction between the left sub-tree and the right sub-tree is arbitrary, but it is a convenient bit of terminology that reflects the way we draw trees. An example of a familiar binary tree can be found in your "family tree". Draw a node with yourself as the root. To the left, draw a "child" node representing your father, and to the right a "child" node representing your mother. In this tree, a node's "children" (in tree terminology) are its parents (in the English-language sense)! Repeat this for each node, as far as you can.

Exercise 16.3: Actually, if you repeat the above process far enough, the structure you get will no longer be a tree. Why?

Describing Binary Trees in Java

As with the other data structures we have discussed previously, we will develop a `WCSBinaryTree` class. The `Node` class we have been using will not suffice here because we need pointers to both the left and right sub-trees. When a `TreeNode` is missing a left or right sub-tree, this will be indicated by a value of `null` for the corresponding reference. Note that in the code for class `TreeNode` below, we have not used the `private` access modifier for the instance variables of each `TreeNode`. By omitting any access modifier, we are using the default access which is access within the package containing the class. On a Mac, default access is access by any class in the same folder. On a UNIX system, default access is access by any class within the same directory. We plan to put our binary tree class in the same package with the `TreeNode` class. As we will see below in the `WCSBinaryTree` class, using default access allows us to write shorter, more efficient code. Since our `TreeNode` class and our `WCSBinaryTree` class work together and never separately, it is okay to give away some of the security provided through the use of the `private` access modifier. Figure 16.4 shows a class definition appropriate for a binary `TreeNode`.

```
public class TreeNode<E> {
    E myData; //reference to the data being organized in the binary tree
    TreeNode<E> myLeft;//reference to left sub-tree
    TreeNode<E> myRight;//reference to right sub-tree
    public TreeNode(E theData) {

       myData = theData;
       myLeft = null;
       myRight = null;
    }//1-parameter constructor
}//class TreeNode
```

Figure 16.4 A class to Represent a Node of a Binary Tree.

We manipulate binary trees in many of the same ways we manipulate linked lists: we construct, traverse, search, add to, and delete from trees. The algorithms for these manipulations must take into account the non-linear nature of the tree. For example, this leads to at least three different, useful ways to traverse a `WCSBinaryTree`. There are also various ways to construct `WCSBinaryTree`s, depending on the kind of relationships between the `TreeNode`s which are reflected by the tree structure. Two of the most important uses of `WCSBinaryTree`s are as search trees and as expression trees. We will discuss the uses, construction, and traversal of search trees in some detail, and talk briefly about the use of expression trees.

Search Trees

The most common use of the binary tree structure is probably in the role of a search tree. A search tree is a `WCSBinaryTree` in which each `TreeNode` has a key value, and in which each `TreeNode` has the property: all the keys of all the `TreeNode`s in the left sub-tree are less than the `TreeNode`'s key, and all the keys of all the `TreeNode`s in the right sub-tree are greater than the `TreeNode`'s key. Figure 16.5 shows three examples of search trees. In one of the examples in this figure, we use English words as our keys. We consider the words to be ordered by the usual alphabetical order. Note that an ordered linked list is a form of "degenerate" search tree, as in the third example in Figure 16.5.

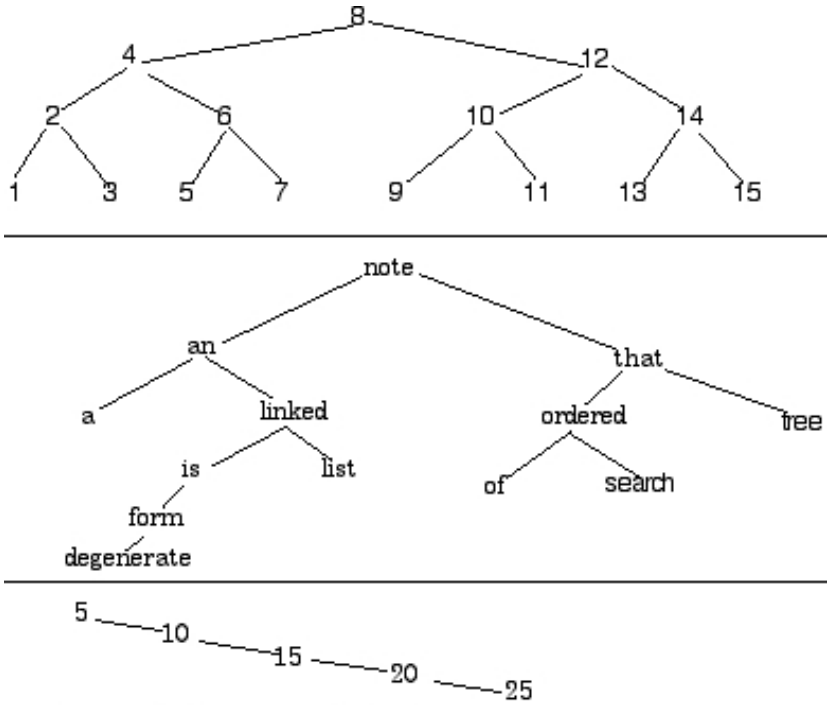


Figure 16.5 Examples of Search Trees

Building Search Trees

Search trees, as their name implies, are useful for searching. That is, if you have a group of data items properly organized in a search tree, and you wish to know if an item with a particular key is in the tree, it turns out to be simple and efficient to search the tree for the item. We will describe the construction of a search tree first, and then show how the search is done. Here is a simple problem to place the construction in a context:

Read a sequence of integers, and construct a search tree with the integers as keys. If an integer is encountered with the same value as an integer already installed in the tree, then ignore it.

The algorithm for building the tree uses a recursive subalgorithm. The idea is as follows. Each integer will be inserted according to the following scheme. We examine the root of the tree. If the root is `null`, we have found the place to insert a new `TreeNode`. Hence, we construct a `TreeNode` and make the root refer to it. If the root is not `null`, we

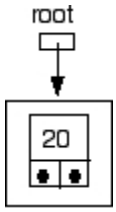
compare the integer being inserted to the integer in the root TreeNode. If the integer being inserted is less than the integer in the root TreeNode, we recursively begin the process again using the left sub-tree. If the integer being inserted is greater than the integer in the root TreeNode, we recursively begin the process again using the right sub-tree. If the integer is equal to the integer in the root TreeNode, just ignore that integer.

Let us illustrate the logic by building a search tree with the following sequence of integers:

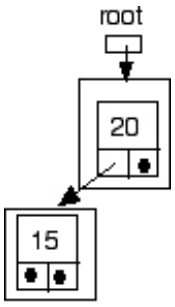
20 15 10 30 25 18 6 28 13

Since the root TreeNode begins as null,

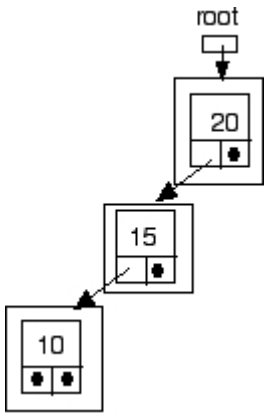
we construct a TreeNode using the value 20 and make the root refer to that node as follows:



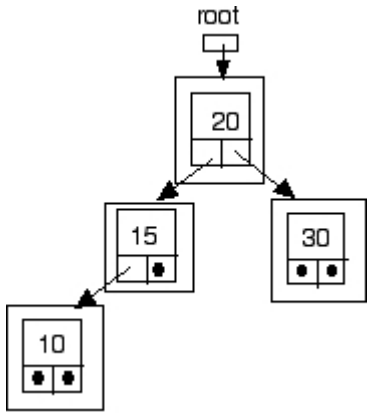
The value 15 is next to be inserted. We examine the root which is not null. Then since 15 is less than 20, we recurse with the left sub-tree. Since the root of the left sub-tree is null, we have found the place to insert 15 in the tree. We construct a TreeNode with 15 and make the root of the left sub-tree refer to the newly constructed node as shown below.



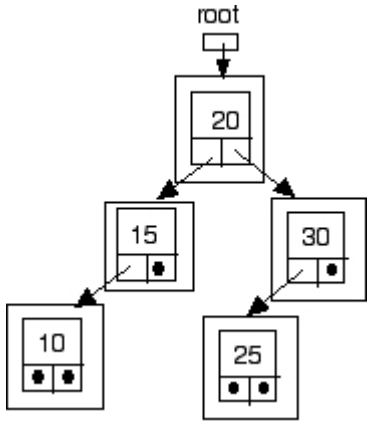
The value 10 is next to be inserted. We examine the root which is not null. Since 10 is less than 20, we recurse with the left sub-tree. Since the root of the left sub-tree is not null, we have not found the place to insert 10. Since 10 is less than 15, we recurse with the root of the left sub-tree off of 15. That sub-tree is null. Hence, we have found the place to insert 10. We construct a TreeNode with 10 and make the root of the left sub-tree off of 15 refer to the newly constructed node as shown below.



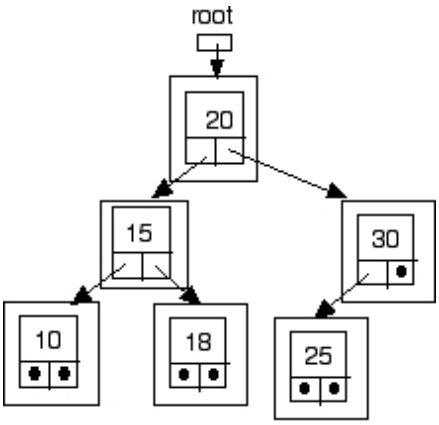
The value 30 is next to be inserted. We examine the root which is not `null`. Since 30 is greater than 20, we recurse with the right sub-tree. Since the root of the right sub-tree is `null`, we have found the place to insert 30. We construct a `TreeNode` with 30 and make the right sub-tree off of 20 point to the newly constructed node as shown below.



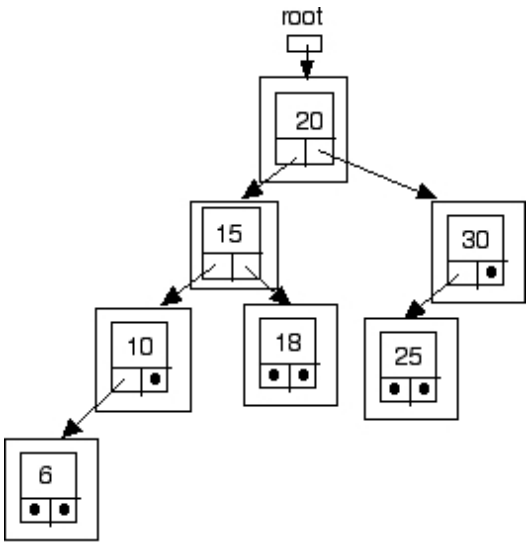
The value 25 is next to be inserted. We examine the root which is not `null`. Since 25 is greater than 20, we recurse with the right sub-tree. Since the root of the right sub-tree is not `null`, we have not found the place to insert 25. Since 25 is less than 30, we recurse with the root of the left sub-tree off of 30. That sub-tree is `null`. Hence, we have found the place to insert 25. We construct a `TreeNode` with 25 and make the root of the left sub-tree off of 30 point to the newly constructed node as shown below.



The value 18 is next to be inserted. We examine the root which is not `null`. Since 18 is less than 20, we recurse with the left sub-tree. Since the root of the left sub-tree is not `null`, we have not found the place to insert 18. Since 18 is greater than 15, we recurse with the right sub-tree off of 15. That sub-tree is `null`. Hence, we have found the place to insert 18. We construct a `TreeNode` with 18 and make the right sub-tree off of 15 point to the newly constructed node as shown below.

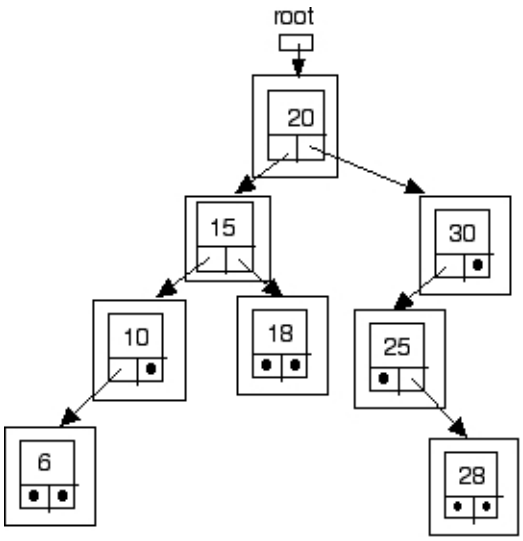


The value 6 is next to be inserted. We examine the root which is not `null`. Since 6 is less than 20, we recurse with the left sub-tree. Since the root of the left sub-tree is not `null`, we have not found the place to insert 6. Since 6 is less than 15, we recurse with the root of the left sub-tree off of 15. Since the root of the root of the left sub-tree off 15 is not `null`, we have not found the place to insert 6. Since 6 is less than 10, we recurse with the root of the left sub-tree off of 10. That sub-tree is `null`. Hence, we have found the place to insert 6. We construct a `TreeNode` with 6 and make the root of the left sub-tree off of 10 point to the newly constructed node as shown below.

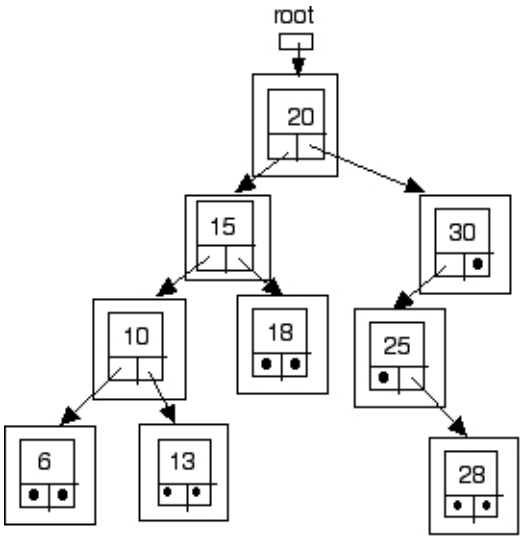


The value 28 is next to be inserted. We examine the root which is not `null`. Since 28 is greater than 20, we recurse with the right sub-tree. Since the root of the right sub-tree is not `null`, we have not found the place to insert 28. Since 28 is less than 30, we

recurse with the root of the left sub-tree off of 30. Since the root of the left sub-tree off 30 is not `null`, we have not found the place to insert 28. Since 28 is greater than 25, we recurse with the right sub-tree off of 25. That sub-tree is `null`. Hence, we have found the place to insert 28. We construct a `TreeNode` with 28 and make the root of the left sub-tree off of 25 point to the newly constructed node as shown below.



The value 13 is next to be inserted. We examine the root which is not `null`. Since 13 is less than 20, we recurse with the left sub-tree. Since the root of the left sub-tree is not `null`, we have not found the place to insert 13. Since 13 is less than 15, we recurse with the root of the left sub-tree off of 15. Since the root of the left sub-tree is not `null`, we have not found the place to insert 13. Since 13 is greater than 10, we recurse with the right sub-tree off of 10. That sub-tree is `null`. Hence, we have found the place to insert 13. We construct a `TreeNode` with 13 and make the right sub-tree off of 10 point to the newly constructed node as shown below.



The tree is complete.

Exercise 16.4: Suppose the input to this algorithm consists of the same 10 numbers as those we traced, but in a different sequence. Then the resulting tree may not look the same. Trace the algorithm with the inputs:

a) 15 10 30 25 18 6 28 13 20
b) 6 10 13 15 18 20 25 28 30
c) 13 28 6 18 25 30 10 15 20
d) 20 30 25 28 15 10 6 13 18

In each case, draw the resulting tree.

Writing a WCSBinaryTree Class

Now, let us start to write our class appropriately named `WCSBinaryTree`. As usual, we are hiding the particular data structure from a user of the class. We use the `private` access modifier for the root of a `WCSBinaryTree`. We have an additional `private` instance variable for a `Comparator<E>` object that may be employed to order the tree in a way that is different from the natural ordering of its data objects provided through the `compareTo` method. There are two constructors whose signatures differ with respect to the existence of a `Comparator<E>` parameter. Both constructors simply assign the value `null` to the `myRoot` and assign the appropriate value to `myComparator`. We begin the class definition as shown below, including the method `myCompare` that selects between using `myComparator` and the `compareTo` method.

```
public class WCSBinaryTree<E extends Comparable<E>> {
    private  TreeNode<E> myRoot; // the TreeNode at the root of the tree
    private Comparator<E> myComparator; // for ordering other than natural ordering
    public WCSBinaryTree () {
         myRoot = null;
         myComparator = null;
    }  //0 parameter constructor
    public WCSBinaryTree (Comparator<E> theComparator) {
         myRoot = null;
         myComparator = theComparator;
    }  //1 parameter constructor
    private int myCompare (E firstE, E secondE){
         if(myComparator == null) {
              return firstE.compareTo(secondE);
         }//no comparator
         return myComparator.compare(firstE, secondE);
    }//myCompare
```

Recall from our discussion of recursive list processing that our methods typically come in pairs, one method being `public` and the second being `private`. The parameter of the `public` method to perform insertion into the tree is simply the object to be inserted. The `public` method invokes the `private` method, using the hidden data item that is the root of the tree. Recall from the discussion above that insertion occurs when the

root or the root of a sub-tree is found to be null. We can write an algorithm to embody this idea as follows. We will use the name "currentRoot" to refer to the root of the sub-tree we are examining at any particular moment.

If currentRoot is null, then

construct a new TreeNode with the object to be inserted and null left and right references and make currentRoot refer to it by returning a reference to the new TreeNode.

Otherwise, if the key in the object to be inserted is before the key in the TreeNode referenced by currentRoot,

begin the insertion process with the left sub-tree and return currentRoot since it should not be changed.

Otherwise, if the key in the object to be inserted is after the key in the TreeNode referenced by currentRoot,

begin the insertion process with the right sub-tree and return currentRoot since it should not be changed.

After having recursed either left or right appropriately or not at all if the key is already in the tree, return the currentRoot.

The main idea in coding this algorithm is that an assignment is made to a null reference. However, as the code searches through the tree to locate the null sub-tree, it must not lose existing links. Hence, the code will reassign already existing references to sub-trees. This seemingly redundant reassigning of references allows the code to be short and elegant.

Let us first consider the public method of the pair. Think about what must happen when the tree is empty. A new TreeNode must be constructed and the root must be assigned to refer to that new TreeNode. Hence, an assignment to the root of the tree appears appropriate. In addition, according to our last paragraph, we will reassign the root whenever a subsequent object is inserted in the tree. Hence, the code below assigns the value returned by the private method to the root, and it should be clear that the private method returns either a reference to the new TreeNode constructed (in the case that this is the first TreeNode inserted) or a reference to the same TreeNode to which it is currently referring (in the case that this in not the first TreeNode inserted).

Analogous to our previous discussion of recursive pairs of methods, the following statements describe the structure of the code.

- The class WCSBinaryTree will have two instance variables that facilitate the insertion process. The first is of type boolean and is named addSuccessful. The second is of class E and is named objectBeingAdded.
- The return type of the public add method is boolean indicating whether the composition of the tree was changed by the actions of the public and private add methods.
- The public add method has one parameter of class E that is a reference to the object being inserted.

- The public add method assigns the value false to addSuccessful and assigns its parameter to objectBeingAdded before the private add method is invoked.
- The root of the tree becomes the argument in the invocation of the private add method.
- The public add method returns the value of addSuccessful that may have been changed by the actions of the private add method.
- The return type of the private add method is TreeNode<E>.

Hence, the code for the instance variables and the public add method is as follows:

```
// additional instance variables of the class
private  boolean  addSuccessful;
private  E  objectBeingAdded;

public boolean add (E  addMe) {
    addSuccessful = false;
    objectBeingAdded = addMe;
    myRoot  = add(myRoot);
    return  addSuccessful;
}// public add
```

Now we consider the code of the private add method. The code must first test to see if the currentRoot is null, and if it is, a new TreeNode is constructed and returned to the calling method. Hence the method begins as follows:

```
private  TreeNode<E>  add(TreeNode<E>  currentRoot) {
    if (currentRoot  ==  null) {
        addSuccessful = true;
        return  new  TreeNode< >(objectBeingAdded);
    }// insertion position found
```

When the currentRoot is not null, a test must be performed to decide whether to continue looking for a null sub-tree to the left or to the right of the currentRoot. We declare a local int variable compareResult to hold the value returned by myCompare since we may need to test it twice. If the key of objectBeingAdded is before the key in the currentRoot's TreeNode, we make the recursive call to start the process again with the left sub-tree. Hence, the code continues as follows:

```
int compareResult = myCompare(objectBeingAdded,currentRoot.myData);
if (compareResult < 0) {
    currentRoot.myLeft = insert (currentRoot.myLeft);
```

The assignment statement in the code above should make sense because it is very similar to the assignment statement in the public method. Remember that the value that is assigned to currentRoot.myLeft will either be a reference to a new TreeNode if this is the place to insert the new TreeNode or a reassignment of its current value.

If the key in objectBeingAdded is not before the key in currentRoot's node, we test to see if it is after and recurse to the right in that case. Hence, the code continues in a similar fashion as shown below.

```
} else if (compareResult > 0) {
    currentRoot.myRight = insert (currentRoot.myRight);
} // else if
```

Our last case is that the key is already in the tree and we have agreed not to insert duplicates. Hence, we do nothing.

The last statement that we need is:

```
return currentRoot;
```

Our private add method must return a reference to a TreeNode. If we have just made a recursive call, it means that we had not found the correct place in the tree. In that case, the value returned to the calling method must be the same value of currentRoot to make the redundant assignment work correctly. You may think of this as rebuilding the current structure of the binary tree.

We have assembled the private add method below in for your convenience.

```
private TreeNode<E> add (TreeNode<E> currentRoot) {
    if (currentRoot == null) {
        addSuccessful = true;
        return new TreeNode< > (objectBeingAdded);
    } // insertion position found
    int compareResult = myCompare(objectBeingAdded,currentRoot.myData);
    if (compareResult < 0) {
        currentRoot.myLeft = insert (currentRoot.myLeft);
    } else if (compareResult > 0) {
        currentRoot.myRight = insert (currentRoot.myRight);
    } // else if
    return currentRoot;
}// private add method
```

Searching a Search Tree

Having built a search tree, we can now search it. The power of the search tree structure arises from the fact that, under certain circumstances, it provides an extremely simple—in fact, almost automatic—implementation of the binary search algorithm (see Chapter 11). Recall that we can search an array for an item using binary search if the array is sorted. However, we cannot use binary search for a linked list, even if the list is sorted, since the only access to list items is sequential. The power of binary search, of course, is its speed. To find an item in a sequence of one million items can take up to one million comparisons if the search is done sequentially. Using binary search, you will never need more than 20 comparisons to find an item in an ordered array of one million items.

To illustrate why search trees can provide some of the power of binary search, and what the necessary circumstances are, consider the search tree in Figure 16.6:

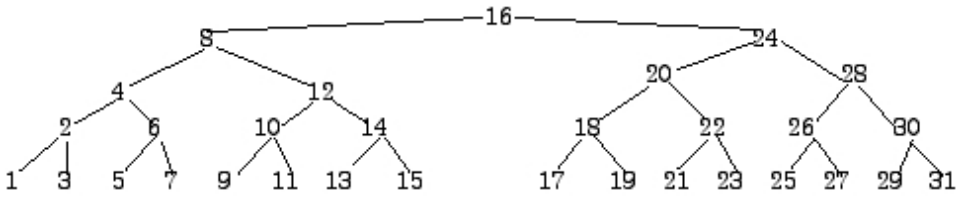


Figure 16.6 A Binary Search Tree

This search tree is nicely symmetric. It contains all the integers between 1 and 31, inclusive. Suppose we do not know that, and we want to find out if the value 18 is in the tree. Initially, we only have access to the root node. We compare our search key (18) to the root key, and discover that the search key is greater. This immediately eliminates half the tree from consideration: we know that, if 18 is in the tree, it will be in the right sub-tree below 16. So we examine the root of this sub-tree, which is 24. The search key is less than this key, so if 18 is anywhere in the tree it will be in the left sub-tree below 24. That sub-tree is rooted at 20, and since 18 is less than 20, we know that if 18 is in the tree it will be in the left sub-tree below 20. We look at the root of this sub-tree, and find that it is 18. Our search is complete.

Each comparison moves us down one level of this tree. At each stage in the search, we can eliminate half of the remaining part of the tree. This is precisely the feature that makes binary search so powerful. Since the depth of the tree in Figure 16.6 is 5, we will never need more than 5 comparisons to either find a key in this tree, or decide that it is not present. However, suppose the same 31 keys are organized into a search tree as in Figure 16.7:

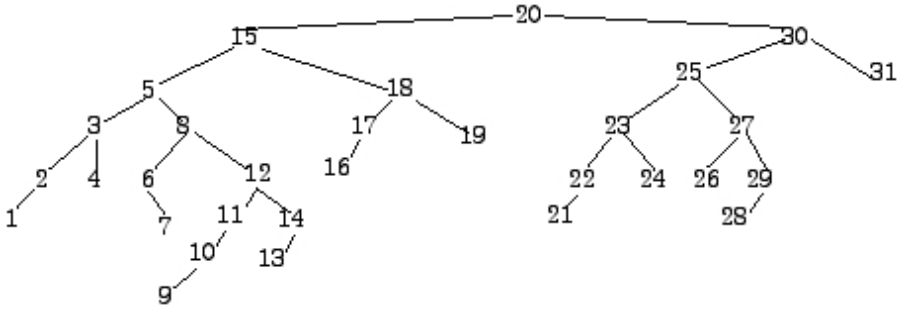


Figure 16.7 Binary Search Tree with Depth 8

This search tree has depth 8. It may take as many as 8 comparisons to find a key, or determine that it is not present. And things can be much worse. Figure 16.8 shows part of a "worst case" search tree for these 31 keys:

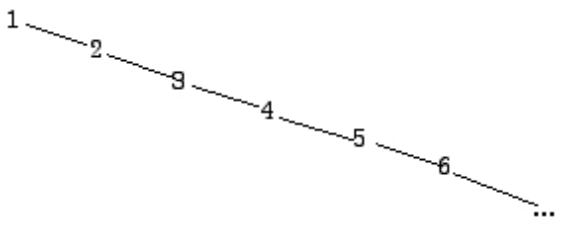


Figure 16.8 The Worst Case is a Binary Tree that is Really an Ordered List

The search tree can be an ordered list, in which case the search becomes equivalent to sequential search. Thus, the efficiency of searching a search tree depends on how well the tree is "balanced". As a result, much effort has been spent by researchers in determining the best way to construct search trees that are balanced, or nearly balanced, regardless of the order in which the data arrives. (Exercise 16.4 shows that, using the algorithm we gave for building trees, the same input values arriving in different orders give rise to trees of very different shapes.)

We will not discuss the tree balancing problem further in this book. It is just an example of the sorts of problems that arise in the study of data structures. Our next order of business is to describe an algorithm, and then write Java code, for the searching process we described above. It is much simpler than tree construction.

The search algorithm is recursive. To begin, we test to see if the current root is `null`. If it is, then the object is not in the tree. If the current root is not `null`, we test to see if the object for which we are searching is `equal` to the `myData` field the current root node. If it is, we have found the object and we return a reference to the `myData` field of the `TreeNode` which is referenced by the current root. If we have not found the object, we need to decide whether to search left or right. Arbitrarily, we test to see whether the object for which we are searching is before the object in the current root node. If it is, we recursively search to the left of the current root node. If it is not, we recursively search to the right of the current root node.

Once again our search algorithm is coded into a pair of methods, one `public` and one `private`. Both methods have return type `E`. When a `null` value is returned, our code must interpret the `null` pointer as indicating that the object is not in the tree. The `public` method simply assigns its parameter to the instance variable of class `E` named `objectSearchedFor` and returns the reference that the `private` method returns to it. The `public` method as shown below.

```
public E search(E findMe) {
     objectSearchedFor = findMe;
     return search(myRoot);
}//public search
```

The `private` method embodies the algorithm described above. If at any point in the recursion, we have a `currentRoot` that is `null`, we return `null` because the object is not in the tree. If the `currentRoot` is not `null`, and we find that the object for which we are searching is equal to the `myData` field of the `TreeNode` referenced by `currentRoot`, we will return the `myData` field of `currentRoot`. If the object for which we are searching is before the `myData` field of the `TreeNode` referenced to by `currentRoot`, we return the result of a recursive call to `search` using the left sub-tree. Finally, if the above three conditions all fail, we must be in the case where the object for which we are searching is after the `myData` field of the `TreeNode` referenced to by `currentRoot`, and we return the result of a recursive call to `search` using the right sub-tree. The `private search` method is displayed below.

```
private E search (TreeNode<E>  currentRoot) {
     if (currentRoot  ==  null) {
          return null;
```

```
        }// object  not  in tree
        if (objectSearchedFor.equals(currentRoot.myData)){
             return currentRoot.myData;
        }// found  the  object
        if ( myCompare (objectSearchedFor, currentRoot.myData) < 0) {
             return  search (currentRoot.myLeft);
        }// search left sub-tree
        return  search (currentRoot.myRight); // search right sub-tree
    }// private search
```

Deleting Items from a Search Tree

Having seen how to build search trees and how to search for items in a search tree, we might next ask how to delete items from a binary search tree. This turns out to be surprisingly complex. We give an introductory discussion of it here, and leave coding the solution as an exercise.

If we want to delete a node from a search tree, part of the problem is to ensure that the remaining portion of the tree still satisfies the definition of a search tree. There are three cases to deal with:

- The `TreeNode` to be deleted has no children, i.e., is a "leaf" node. This case is the simplest, since we can simply make the link from the `TreeNode`'s parent `null`.
- The `TreeNode` to be deleted has only one child. In this case, we simply "promote" the child, i.e., make its grandparent into its parent, by assigning to the link from the parent the value of the link in the `TreeNode` we want to delete.
- The `TreeNode` to be deleted has two children. This case is more difficult. We will explain below an example algorithm for deleting such a `TreeNode`.

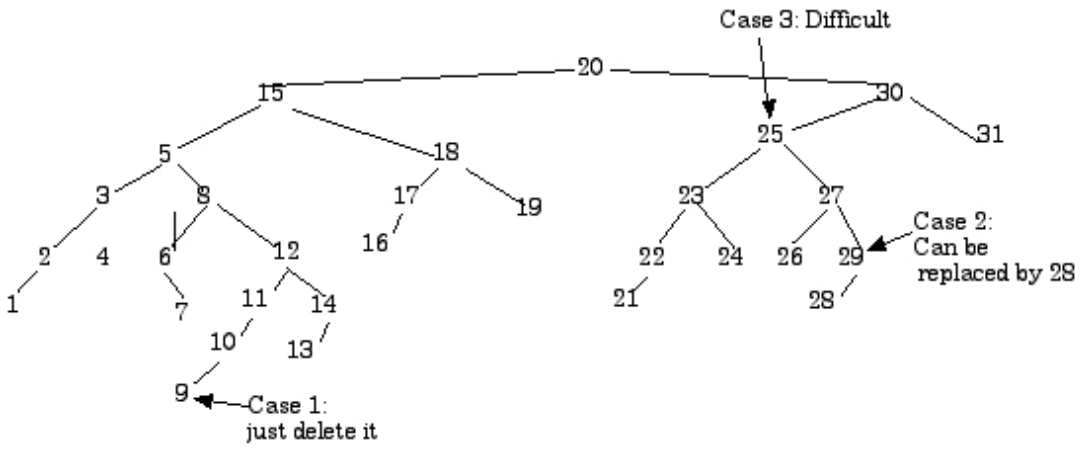


Figure 16.9 Three Cases for Deletion of a Binary TreeNode

Exercise 16.5: Redraw the tree in Figure 16.9 after deleting the TreeNode containing 25, in such a way that it remains a search tree, and no part of the tree changes except the sub-tree under the node containing 25.

One way to think about deleting a TreeNode with two children is that every value in the TreeNode's right sub-tree is greater than every value in the TreeNode's left sub-tree. In particular, every value in the right sub-tree is greater than the largest value in the left sub-tree. Hence, our strategy could be to attach the right sub-tree to the rightmost node of the left sub-tree and "promote" the top TreeNode of the left sub-tree to take the place of the TreeNode being deleted.

Exercise 16.6: It is similarly true that every value in the TreeNode's left sub-tree is less than the leftmost value in the TreeNode's right sub-tree. Describe the strategy for deletion that follows from this fact and is similar to the strategy described in the last paragraph.

Exercise 16.7: Redraw the tree after deleting the TreeNode containing 25 according to the strategy described above where the top TreeNode of the left sub-tree takes the place of the TreeNode being deleted.

Exercise 16.8: Below is a public delete method for the WCSBinaryTree class that is passed an object to delete and that returns a boolean value. The method returns true if the object is successfully deleted from the tree. The method returns false in the case that the object is not found and hence not deleted from the tree. This method invokes a private delete method of the WCSBinaryTree class that is recursive and does the work to delete the TreeNode. Note that the private method is passed a reference to the root of the tree. Also note that the WCSBinaryTree class is assumed to have a boolean instance variable named deleteSuccessful and an instance variable of class E named objectToDelete. Write the corresponding private delete method for the WCSBinaryTree class.

```
public  boolean  delete(E  deleteMe) {
      objectToDelete = deleteMe;
      deleteSuccessful = false;
      myRoot = delete (myRoot);
      return  deleteSuccessful;
}// public  delete
```

Traversing Binary Trees

By traversing a tree we mean visiting each of its nodes exactly once. The expression "visiting a node" can include applying some simple operation to the TreeNode, such as concatenating the result of the method toString on its myData field to a temporary

String. We are going to describe three different algorithms for traversing binary trees. Each of these algorithms is recursive, each can be applied to any kind of binary tree (including search trees), and each has its uses under certain circumstances. The recursive descriptions of the algorithms are remarkably simple and remarkably similar.

The first algorithm is known as pre-order traversal. In this strategy, each TreeNode is visited before any of the TreeNodes of its left and right sub-trees. The description is:

If the currentRoot is not null,

a) Visit the TreeNode to which currentRoot refers.

b) Apply this algorithm to the left sub-tree.

c) Apply this algorithm to the right sub-tree.

To illustrate this method, we will trace it on the tree shown in Figure 16.10.

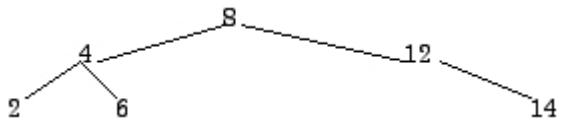


Figure 16.10 Sample WCSBinaryTree for tracing pre-order traversal

currentRoot(#1) points to the TreeNode containing 8 initially. Since that reference is not null, we have three steps to perform:

1a) Visit the TreeNode containing 8.

1b) Apply this algorithm to its left sub-tree (whose root node contains 4).

1c) Apply this algorithm to its right sub-tree (whose root node contains 12)

Hence, we visit the TreeNode containing 8 and begin the algorithm with its left sub-tree. We will need to remember to apply the algorithm to its right sub-tree (step 1c) after we complete the algorithm for its left sub-tree (step 1b). Hence, step 1c is suspended.

currentRoot(#2) points to the TreeNode containing 4. Since that reference is not null, we have three steps to perform:

2a) Visit the TreeNode containing 4.

2b) Apply this algorithm to its left sub-tree (whose root node contains 2).

2c) Apply this algorithm to its right sub-tree (whose root node contains 6)

Hence, we visit the TreeNode containing 4 and begin the algorithm with its left sub-tree. We will need to remember to apply the algorithm to its right sub-tree (step 2c) after we complete the algorithm for its left sub-tree (step 2b). At this point steps 1c and 2c are suspended.

currentRoot(#3) points to the TreeNode containing 2. Since that reference is not null, we have three steps to perform:

3a) Visit the TreeNode containing 2.

3b) Apply this algorithm to its left sub-tree (whose root node is null).

3c) Apply this algorithm to its right sub-tree (whose root node is null)

Hence, we visit the TreeNode containing 2 and begin the algorithm with its left sub-tree. We will need to remember to apply the algorithm to its right sub-tree (step 3c) after we complete the algorithm for its left sub-tree (step 3b). At this point steps 1c, 2c, and 3c are suspended.

currentRoot(#4) is a null TreeNode. Hence, we complete the left sub-tree off the node containing 2 (step 3b) without any more work. currentRoot(#4) no longer exists. It is now time to work on the suspended step 3c, the right sub-tree off the node containing 2.

currentRoot(#5) is a null TreeNode. Hence, we complete the right sub-tree off the node containing 2 (step 3c) without any more work. currentRoot(#5) no longer exists. Note that we have completed steps 3a, 3b, and 3c, and currentRoot (#3) no longer exists. It is now time to work on the suspended step 2c, the right sub-tree off the node containing 4.

currentRoot(#6) refers to the TreeNode containing 6. Since that reference is not null, we have three steps to perform:

4a) Visit the TreeNode containing 6.

4b) Apply this algorithm to its left sub-tree (whose root node is null).

4c) Apply this algorithm to its right sub-tree(whose root node is null)

Hence, we visit the TreeNode containing 6 and begin the algorithm with its left sub-tree. We will need to remember to apply the algorithm to its right sub-tree (step 4c) after we complete the algorithm for its left sub-tree (step 4b). At this point steps 1c, 2c, and 4c are suspended.

currentRoot(#7) is a null TreeNode. Hence, we complete the left sub-tree off the node containing 6 (step 4b) without any more work. currentRoot(#7) no longer exists. It is now time to work on the suspended step 4c, the right sub-tree off the node containing 6.

currentRoot(#8) is a null TreeNode. Hence, we complete the right sub-tree off the node containing 6 (step 4c) without any more work. currentRoot(#8) no longer exists. Steps 4a, 4b, and 4c have been completed, and currentRoot(#6) no longer exists.

Step 2c has also been completed since we have completed the right sub-tree off the node containing 4, and currentRoot(#2) no longer exists.

Step 1b has also been completed since we are done with the left sub-tree off the node containing 8. It is now time to work on the suspended step 1c, the right sub-tree off the node containing 8.

currentRoot(#9) refers to the TreeNode containing 12. Since that reference is not null, we have three steps to perform:

5a) Visit the TreeNode containing 12.

5b) Apply this algorithm to the left sub-tree (whose root node is null).

5c) Apply this algorithm to the right sub-tree (whose root node contains 14)

Hence, we visit the TreeNode containing 12 and begin the algorithm with its left sub-tree. We will need to remember to apply the algorithm to its right sub-tree (step 5c)

after we complete the algorithm for its left sub-tree (step 5b). At this point step 5c is suspended.

currentRoot(#10) is a null TreeNode. Hence, we complete the left sub-tree off the node containing 12 (step 5b) without any more work. currentRoot(#10) no longer exists. It is now time to work on the suspended step 5c, the right sub-tree off the node containing 12.

currentRoot(#11) refers to the TreeNode containing 14. Since that reference is not null, we have three steps to perform:

6a) Visit the TreeNode containing 14.

6b) Apply this algorithm to the left sub-tree (whose root node is null).

6c) Apply this algorithm to the right sub-tree(whose root node is null)

Hence, we visit the TreeNode containing 14 and begin the algorithm with its left sub-tree. We will need to remember to apply the algorithm to its right sub-tree (step 6c) after we complete the algorithm for its left sub-tree (step 6b). At this point step 6c is suspended.

currentRoot(#12) is a null TreeNode. Hence, we complete the left sub-tree off the node containing 14 (step 6b) without any more work. currentRoot(#12) no longer exists. It is now time to work on the suspended step 6c.

currentRoot(#13) is a null TreeNode. Hence, we complete the right sub-tree off the node containing 14 (step 6c) without any more work. currentRoot(#13) no longer exists.

We have also just completed the right sub-tree off the node containing 12, finishing step 5c. We have finally completed step 1c, the right sub-tree off the node containing 8, and currentRoot(#1) no longer exists.

Note that the order in which we visited the nodes was:

8, 4, 2, 6, 12, 14

Exercise 16.9: Traverse each of the trees in Figure 16.5 in pre-order. Indicate the order in which the nodes are visited.

Figure 16.11 shows a pair of methods for the WCSBinaryTree class that implement a toStringPreOrder method for the class using pre-order traversal.

```
public  String  toStringPreOrder() {
     return toStringPreOrder (myRoot);
} // public  toStringPreOrder

private String toStringPreOrder (TreeNode<E>  currentRoot) {
     if (currentRoot != null ) {
         return currentRoot.myData.toString() + "\n" +
           toStringPreOrder (currentRoot.myLeft) +
           toStringPreOrder (currentRoot.myRight);
     }// if
     return "";
} // private  toStringPreOrder
```

Figure 16.11 Methods to implement toStringPreOrder

A second tree traversal strategy is called in-order traversal. In this strategy, each `TreeNode` is visited after all of the nodes of its left sub-tree, but before any of the nodes of its right sub-tree. The description is:

If the `currentRoot` is not `null`,

a) Apply this algorithm to the left sub-tree.

b) Visit the `TreeNode` to which `currentRoot` refers.

c) Apply this algorithm to the right sub-tree.

Exercise 16.10: Trace the algorithm for in-order traversal on the tree shown in Figure 16.10. Model your trace on our trace above.

Exercise 16.11: Trace the algorithm for in-order traversal on each of the trees in Figure 16.5. Indicate the order in which the nodes are visited. Model your trace on our trace above.

Exercise 16.12: Write `public` and `private` `toStringInOrder` methods for the `WCSBinaryTree` class modeled on the methods in Figure 16.11.

The third tree traversal strategy is called post-order traversal. In this method, each node is visited only after all of the nodes of both of its sub_trees are visited and the nodes of its left sub-tree are all visited before the nodes of its right sub-tree.

Exercise 16.13: Write a description of the algorithm for post-order tree traversal using the descriptions above for pre-order and in-order traversal as models.

Exercise 16.14: Trace your algorithm for post-order traversal on the tree shown in Figure 16.10. Model your trace on our trace above.

Exercise 16.15: Trace the algorithm for post-order traversal on each of the trees in Figure 16.5. Indicate the order in which the nodes are visited. Model your trace on our trace above.

Exercise 16.16: Write `public` and `private` `toStringPostOrder` methods for the `WCSBinaryTree` class modeled on the methods in Figure 16.11.

Expression Trees

An expression tree is a binary tree whose nodes consist of either operators or operands. The operators can be arithmetic, or relational, but they must be either unary or binary operators. That is, they must take either one or two operands. The operands of an expression tree are either values or expression sub-trees. A `TreeNode` that is a value can have no children. A `TreeNode` that is a unary operator must have exactly one child, and a `TreeNode` that is a binary operator must have two children.

Expression trees are useful as a way of writing expressions in parenthesis–free, unambiguous form, making evident the precedence of the operators. For example, consider the expressions and corresponding trees in Figure 16.12:

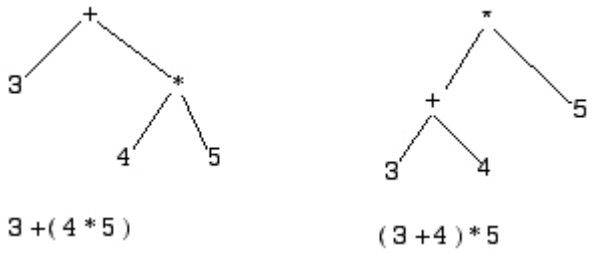


Figure 16.12 Examples of Expression Trees and Corresponding Expressions

In the first expression, the operands of the '+' are its children: the value 3, and its right sub-tree. This right sub-tree itself consists of an operator with two children, both of which are values. This means "add the value 3 to the value of the expression represented by the right sub-tree." We can evaluate the expression tree "from bottom to top", which is equivalent to evaluating the expression itself from innermost parentheses outward. The second tree in Figure 16.14 describes the product of an expression represented by a sub-tree (which is 3 + 4) and 5.

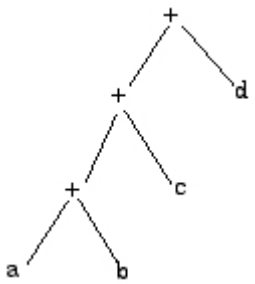
In the absence of parentheses, we can apply Java's rules of precedence to draw the proper tree. For example, the expression

a + b + c + d

is interpreted as

((a + b) + c) + d

because of the left–to–right rule of evaluation. Consequently, it has the expression tree



Exercise 16.17: Draw expression trees for the following un–parenthesized expressions, using the precedence rules for Java:

a) 3 * 4 – 5 * 6
b) 3 + 4 + 5 + 6
c) 3 – 4 / 5 * 6
d) –6 * 3 + 7

Expression trees can be constructed for expressions of any desired level of complexity. The depth of the tree corresponds to the number of levels of operator precedence in the expression. Consider the Java arithmetic expression

```
(a + (b * c)) / (d - (e * (f + g)))
```

We can form an expression tree for this expression as in Figure 16.13:

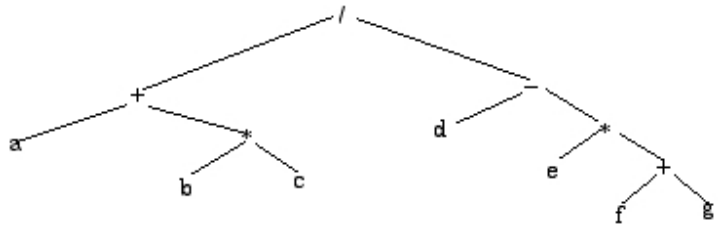


Figure 16.13 Expression Tree for `(a + (b * c)) / (d - (e * (f + g)))`

Another example is shown in Figure 16.14, which illustrates an expression tree for a complex conditional expression:

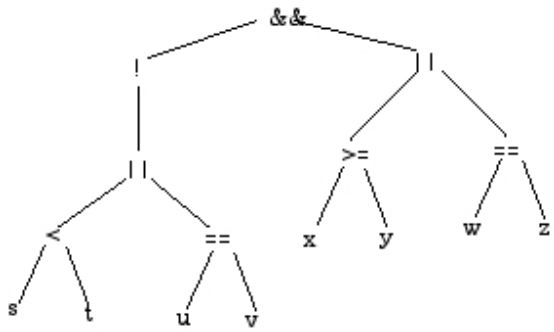


Figure 16.14 Expression Tree for `(! ((s < t) || (u == v))) && ((x >= y) || (w == z))`

It turns out that expression trees are very useful, because it is easy to evaluate an expression once its expression tree has been constructed. Thus, one of the functions of a compiler is to produce expression trees (or equivalent data structures of some sort) for the expressions present in the program being compiled. The construction of an expression tree is complex, and we will not go into it here. However, suppose an expression has been translated into an expression tree. Let us see what happens when we traverse that tree in post-order.

We will start with the two trees in Figure 16.12. Traversing each of them in post-order according to the algorithm given above yields the sequences

```
3 4 5 * +
```

and

```
3 4 + 5 *
```

respectively. Some readers may recognize these forms as the postfix notation (also called reverse Polish notation) for the original expressions. Postfix notation is an unambiguous, parenthesis–free way to write expressions. The postfix form of an expression is useful because it can be evaluated directly, without precedence rules, according to

the following scheme: scan the expression from left to right. When you encounter a value, leave it there. When you encounter a unary operator (one which only takes one operand) erase both the operator and the previous value, replacing them with the result of applying the operator to the value. When you encounter a binary operator, erase the operator and the previous two values, replacing them with the result of applying the operator to those values. For example, Figure 16.15 shows an evaluation of "`3 4 5 * +`", where highlighting is used to indicate which symbol in the expression is being scanned.

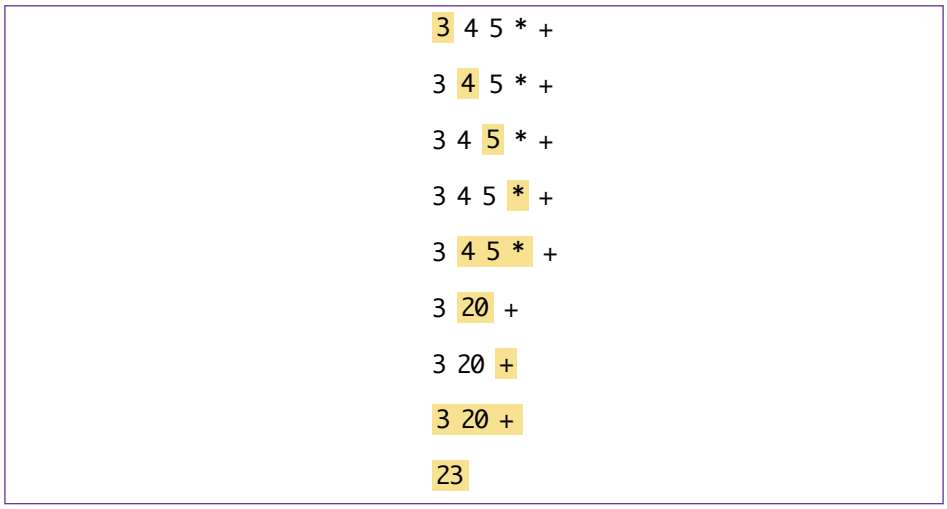


Figure 16.15 Evaluation of a Postfix Expression

An expression can be converted to postfix notation by writing its expression tree, and then traversing that tree in post-order. This process can be applied to expression trees of any complexity. Thus we can convert the expression tree of Figure 16.13 into a postfix expression by post-order traversal, getting

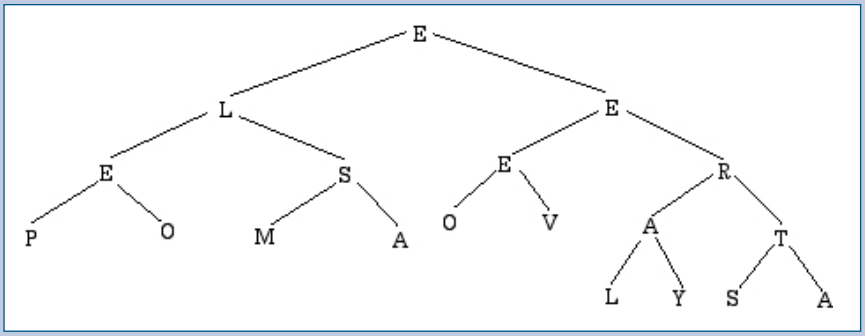
```
a b c * + d e f g + * – /
```

Exercise 16.18: Convert the expression in Figure 16.14 into postfix notation by traversing the corresponding expression tree in post-order. Incidentally, if you traverse an expression tree in pre-order, the resulting expression is called the prefix notation (or Polish notation) form of the expression. It is also parenthesis-free and unambiguous, but the rule for evaluating prefix expressions is less convenient than that for postfix expressions. If you traverse an expression tree in in-order, you will get an expression that looks like the usual algebraic notation for expressions, but without parentheses. Unlike prefix and postfix forms, the algebraic form needs precedence rules and parentheses to be unambiguous.

Exercise 16.19: Write a recursive method that is passed a reference to the root of a binary tree, and returns an integer that is the depth of the tree. If the reference passed is null, then the returned value should be zero. If the reference is not null, but the root has no children, the depth should be one, and so on.

Exercise 16.20: The algorithm steps for pre-order, in-order, and post-order traversal of a binary tree consist of the same three steps, in different orders. There are six possible orders in which to arrange any three things. Thus there are three other ways you could arrange the same algorithm steps. Describe the resulting tree traversals.

Exercise 16.21: Given the following data in a binary tree:



Show the sequence of keys produced when the tree is traversed

a) Pre-order

b) In-order

c) Post-order

Exercise 16.22: Write a method for the `WCSBinaryTree` class that uses a temporary file to delete each node of specified key from a tree. The method should traverse the tree, writing out the `myData` field of each node to the file, except for the nodes to be deleted. Then the program should read back the data from the file and rebuild the tree, following the usual strategy for constructing a search tree. Note that the strategy of tree traversal is important. The three methods we have seen will write the data out in three different orders, which will affect the balance of the new tree to be reconstructed. Which traversal method will be best?

Exercise 16.23: Complete the class named WCSBinaryTree<E extends Comparable<E>> that implements a binary tree of items.

Your WCSBinaryTree<E extends Comparable<E>> class should have at least eight private instance variables. The first instance variable is of class TreeNode<E> and is a reference to the root of the tree. The second instance variable is of class Comparator<E> and is used to maintain the tree in an order specified by the Comparator. The third instance variable keeps a count of items in the list. The count should be updated every time an add or delete operation is successful. The remaining five instance variables are useful during the processes of adding, searching, and deleting from the binary tree. They are:

```
private boolean addSuccessful;
private boolean deleteSuccessful;
private E objectBeingAdded;
private E objectBeingSearchedFor;
private E objectBeingDeleted;
```

One constructor of class WCSBinaryTree <E extends Comparable<E>> will have one parameter that is a reference to a Comparator<E> object. The other constructor will have no parameters.

Your class WCSBinaryTree <E extends Comparable<E>> should have at least the following instance methods:

```
public boolean add(E addMe)
private TreeNode<E> add(TreeNode<E> currentRoot)
public E search(E findMe)
private E search(TreeNode<E> currentRoot)
public boolean delete(E deleteMe)
private TreeNode<E> delete(TreeNode<E> currentRoot)
public boolean isEmpty()
public int getCount()
private int myCompare(E firstItem, E secondItem)
public String saveToFile(String fileName)
private void saveToFile(TreeNode<E> currentRoot)
public String toString()
private String toString(TreeNode<E> currentRoot)
```

You will need a TreeNode<E> class to work in conjunction with your WCSBinaryTree <E extends Comparable<E>> class.

Exercise 16.24: Extension of Exercise 15.17

Revise your class named AddressInfoCollection so that it relies on two WCSBinaryTree<E extends Comparable<E>>s to organize the AddressInfo objects. Test to be sure that your Address Database application works correctly after this change. More specifically:

1) Your class AddressInfoCollection will have two private instance variables that are references to objects of class WCSBinaryTree <AddressInfo> . One WCSBinaryTree <AddressInfo> will be constructed with zero parameter constructor, and hence be ordered by the natural ordering of AddressInfo objects. The second will be constructed with the AddressComp object of class AddressInfo as an argument to the constructor, and hence, be ordered by address.
2) Class AddressInfoCollection will have at least the following methods:
   a) public AddressInfo search(AddressInfo)—This search method will return the value returned by the search method invoked on the WCSBinaryTree ordered by the natural ordering of AddressInfo objects.
   b) public boolean add(AddressInfo)—This add method will invoke the search method described above to check for a duplicate record. If a duplicate is found, false is returned. If no duplicate is found, this insert method will return true only if the attempts to insert the record into both WCSBinaryTree's are successful. It will return false otherwise.
   c) public boolean delete(AddressInfo)—This delete method will invoke the search method described above and assign the returned value to a local AddressInfo reference for the found object. If the returned value is null, the delete method should return false. If the returned value is not null, delete should return true only if the attempts to delete the found reference from both WCSBinaryTree's are successful. The delete method should return false otherwise.
   d) public String toStringAlphabetical()—This method returns the value returned by invoking toString() on the WCSBinaryTree<AddressInfo> that is ordered by the natural ordering of AddressInfo objects.
   e) public String toStringByAddress()—This method returns the value returned by invoking toString() on the WCSBinaryTree<AddressInfo> that is ordered by the AddressComp object of class AddressInfo.
   f) public String saveToFile(String)—This method returns the String returned by invoking saveToFile on the WCSBinaryTree<AddressInfo> that is ordered by the natural ordering of AddressInfo objects with argument equal to the String parameter that is a file name.
   g) public String loadFromFile(String)—This method will be similar to the method loadFromFile in the class List<E extends Comparable<E>> class from Chapter 13. However, in this case, we do not want to invoke loadFromFile on each of the two WCSBinaryTrees. If we did that, we would have two sets of the same data objects in memory that were read from the file. Instead, write loadFromFile in the class AddressInfoCollection so that each object comes just once from the file and is inserted into both WCSBinaryTrees using the add method described above.

Exercise 16.25: Extension of Exercise 15.18

Revise your class named BirthInfoCollection so that it relies on two WCSBinaryTree<E extends Comparable<E>>s to organize the BirthInfo objects. Test to be sure that your Birthday Database application works correctly after this change. More specifically:

3) Your class BirthInfoCollection will have two private instance variables that are references to objects of class WCSBinaryTree<BirthInfo>. One WCSBinaryTree<BirthInfo> will be constructed with zero parameter constructor, and hence be ordered by the natural ordering of BirthInfo objects. The second will be constructed with the ChronComp object of class BirthInfo as an argument to the constructor, and hence, be ordered chronologically.
4) Class BirthInfoCollection will have at least the following methods:
   a) public BirthInfo search(BirthInfo)—This search method will return the value returned by the search method invoked on the WCSBinaryTree ordered by the natural ordering of BirthInfo objects.
   b) public boolean add(BirthInfo)—This add method will invoke the search method described above to check for a duplicate record. If a duplicate is found, false is returned. If no duplicate is found, this insert method will return true only if the attempts to insert the record into both WCSBinaryTree's are successful. It will return false otherwise.
   c) public boolean delete(BirthInfo)—This delete method will invoke the search method described above and assign the returned value to a local BirthInfo reference for the found object. If the returned value is null, the delete method should return false. If the returned value is not null, delete should return true only if the attempts to delete the found reference from both WCSBinaryTree's are successful. The delete method should return false otherwise.
   d) public String toStringAlphabetical()—This method returns the value returned by invoking toString() on the WCSBinaryTree<BirthInfo> that is ordered by the natural ordering of BirthInfo objects.
   e) public String toStringChronological()—This method returns the value returned by invoking toString() on the WCSBinaryTree< BirthInfo > that is ordered by the ChronComp object of class BirthInfo.
   f) public String saveToFile(String)—This method returns the String returned by invoking saveToFile on the WCSBinaryTree<BirthInfo> that is ordered by the natural ordering of BirthInfo objects with argument equal to the String parameter that is a file name.
   g) public String loadFromFile(String)—This method will be similar to the method loadFromFile in the class List<E extends Comparable<E>> class from Chapter 13. However, in this case, we do not want to invoke loadFromFile on each of the two WCSBinaryTrees. If we did that, we would have two sets of the same data objects in memory that were read from the file. Instead, write loadFromFile in the class BirthInfoCollection so that each object comes just once from the file and is inserted into both WCSBinaryTrees using the add method described above.

The Class TreeSet<T> of the Java Collections API

The Java API provides us with a `TreeSet<E>` class that is similar to our class `WCSBinaryTree<E>`. `TreeSet<E>` has four constructors:

1) `TreeSet()`—Constructs a new, empty tree set, sorted according to the natural ordering of its elements.
2) `TreeSet(Collection<? extends E> c)`—Constructs a new tree set containing the elements in the specified collection, sorted according to the natural ordering of its elements.
3) `TreeSet(Comparator<? super E> comparator)`—Constructs a new, empty tree set, sorted according to the specified comparator.
4) `TreeSet(SortedSet<E> s)`—Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

The class `TreeSet<E>` has a variety of methods. A complete list can be found in the Java API documentation. A selection of them that are similar to the methods of class `WCSBinaryTree` are as follows:

- `boolean add(E e)`—Adds the specified element to this set if it is not already present.
- `boolean addAll(Collection <? extends E> c)`—Adds all of the elements in the specified collection to this set.
- `boolean contains(Object o)`—Returns `true` if this set contains the specified element.
- `boolean isEmpty()`—Returns `true` if this set contains no elements.
- `Iterator<E> iterator()`—Returns an iterator over the elements in this set in ascending order.
- `boolean remove(Object o)`—Removes the specified element from this set if it is present.

Note that there is no `search` method in class `TreeSet<E>` that is similar to the `search` method we wrote for class `WCSBinaryTree<E>`. If we want the same functionality, we need a way to return a reference to the `E` in the `TreeSet<E>` that matches the `E` for whom we are searching. The `iterator()` method is the key to producing this functionality. The `iterator()` method returns an `Iterator<E>` object. The abstract methods in the `Iterator<E>` interface are:

- `boolean hasNext()`—Returns `true` if the iteration has more elements.
- `E next()`—Returns the next element in the iteration.

Let us look at code that uses produces an `Iterator<E>` and then uses it to examine each `E` in a `TreeSet<E>`. We will use class `PhoneInfo` from Chapter 13.

We can declare and construct a `TreeSet<PhoneInfo>` named `alphaTreeSet` that is ordered by the `compareTo` method in class `PhoneInfo` as follows:

```
TreeSet<PhoneInfo> alphaTreeSet = new TreeSet<>();
```

and add PhoneInfo objects to the alphaTreeSet as usual. We use the iterator method with the alphaTreeSet as invoking object to produce an Iterator<PhoneInfo> object named it as follows:

```
Iterator< PhoneInfo > it = alphaTreeSet.iterator();
```

We use a while loop in conjunction with the methods hasNext() and next() of the Iterator<PhoneInfo> to examine each PhoneInfo object in the TreeSet as follows:

```
while (it.hasNext()){
    PhoneInfo current = it.next();
    //do whatever is required with current here
}//while
```

Hence we could write a search method to work in a context where alphaTreeSet is known as follows:

```
public PhoneInfo search(PhoneInfo findMe){
     if(!alphaTreeSet.contains(findMe)){
          return null;
     }//no matching PhoneInfo object in the TreeSet
     Iterator< PhoneInfo > it = alphaTreeSet.iterator();
     while (it.hasNext()){
          PhoneInfo current = it.next();
          if(current.equals(findMe)){
                return current;
          }//if found a match
     }//while
     return null;
}//search
```

The pattern of generating an Iterator<E> and using it in conjunction with a while loop and the hasNext() and next() methods was so commonly used that a short form called the *enhanced for loop* was introduced into Java with version 1.5. Using an enhanced for loop, we rewrite the search method above as follows:

```
public PhoneInfo search(PhoneInfo findMe){
     if(!alphaTreeSet.contains(findMe)){
          return null;
     }//no matching PhoneInfo object in the TreeSet
     for (PhoneInfo current : alphaTreeSet) {
          if(current.equals(findMe)){
                return current;
          }//if found a match
     }//for
     return null;
}//search
```

The `Iterator<PhoneInfo>` is working behind the scenes, but is not obvious from the code. We can think of the enhanced `for` loop as an abstraction that allows us to iterate through the elements in a data structure without needing to know how to get from one to the next.

Exercise 16.26: Extension of Exercise 16.24 Revise your class named `AddressInfoCollection` so that it employs two instance variables of class `java.util.TreeSet<AddressInfo>` to organize the `AddressInfo` objects. Your **Database** class should need no revision. More specifically:

1) Your class `AddressInfoCollection` will have two private instance variables that are references to objects of class `TreeSet<AddressInfo>`. One `TreeSet<AddressInfo>` will be constructed with no arguments to the constructor. The second will be constructed with the `public static final Comparator` object of class `AddressInfo` as an argument to the constructor.
2) The methods of class `AddressInfoCollection` are as described below:
   a) `public AddressInfo search(AddressInfo)`—The `search` method will invoke the `contains` method on the `TreeSet` ordered by the natural ordering of its `Comparable` objects. If the `contains` method returns `false`, `search` should return `null`. If the `contains` method returns `true`, either use an enhanced `for` loop or invoke the `iterator` method on the `TreeSet` ordered by the natural ordering of its `Comparable` objects to get an `Iterator` object for that `TreeSet`, and then use a `while` loop with the `Iterator` object to find the reference, and return it.
   b) `public boolean add(AddressInfo)`—The `add` method will invoke the `contains` method on the `TreeSet` ordered by the natural ordering of its `Comparable` objects. If the `contains` method returns `true`, the `add` method should return `false`. If the `contains` method returns `false`, the `add` method will return `true` only if the attempts to `add` the record into both `TreeSet`'s are successful. It will return `false` otherwise.
   c) `public boolean delete(AddressInfo)`—The `delete` method will invoke the `search` method described above and assign the returned value to a local `AddressInfo` reference for the `found` object. If the returned value is `null`, the `delete` method should return `false`. If the returned value is not `null`, `delete` should return `true` only if the attempts to remove the `found` reference from both `TreeSet`'s are successful. The `delete` method should return `false` otherwise.
   d) `public String toStringAlphabetical()`—This method returns the `String` returned by invoking `toString()` on the `TreeSet` ordered by the natural ordering of its `Comparable` objects.
   e) `public String toStringByAddress()`—This method returns the `String` returned by invoking `toString()` on the `TreeSet` ordered by the `public static final Comparator` object of class `AddressInfo`.

(continues)

Exercise 16.26, continued

f) `public String saveToFile(String)`—This method looks very much like the method `saveToFile` from Chapter 13. Instead of a simple `for` loop iterating through the possible array indices, you will need either an enhanced `for` loop or an `Iterator` object working with a `while` loop. You may invoke the `iterator` method on the `TreeSet` ordered by the natural ordering of its `Comparable` objects to get an `Iterator` object for that `TreeSet`, and then use a `while` loop with the `Iterator` to get a reference to each record in the `TreeSet` and write each object to the `ObjectOutputStream`.

g) `public String loadFromFile(String)`—This method will be similar to the method `loadFromFile` in the class `List<E>` from Chapter 13. However, in this case, we do not want to invoke `loadFromFile` on each of the two `TreeSets`. If we did that, we would have two sets of the same data objects in memory that were read from the file. Instead, write `loadFromFile` in the class `AddressInfoCollection` so that each object comes just once from the file and is inserted into both `TreeSet`'s using the `add` method described above.

Exercise 16.27: Extension of Exercise 16.25 Revise your class `BirthInfoCollection` so that it employs two instance variables of `class java.util.TreeSet<BirthInfo>`. Your **Database** class should need no revision.

More specifically:

1) Your class `BirthInfoCollection` will have two private instance variables that are references to objects of class `TreeSet<BirthInfo>`. One `TreeSet<BirthInfo>` will be constructed with the zero parameter constructor. The second will be constructed with the `public static final Comparator` object of class `BirthInfo` as the single argument to the constructor.

2) The methods of class `BirthInfoCollection` are as described below:

   a) `public BirthInfo search(BirthInfo)`—The `search` method will invoke the `contains` method on the `TreeSet` ordered by the natural ordering of its `Comparable` objects. If the `contains` method returns `false`, `search` should return `null`. If the `contains` method returns `true`, either use an enhanced `for` loop or invoke the `iterator` method on the `TreeSet` ordered by the natural ordering of its `Comparable` objects to get an `Iterator` object for that `TreeSet`, and then use a `while` loop with the `Iterator` object to find the reference, and return it.

   b) `public boolean add(BirthInfo)`—The `add` method will invoke the `contains` method on the `TreeSet` ordered by the natural ordering of its `Comparable` objects. If the `contains` method returns `true`, the `add` method should return `false`. If the `contains` method returns `false`, the `add` method will return `true` only if the attempts to `add` the record into both `TreeSet`'s are successful. It will return `false` otherwise.

(continues)

Exercise 16.27, continued

c) `public boolean delete(BirthInfo)`—The `delete` method will invoke the `search` method described above and assign the returned value to a local `BirthInfo` reference for the `found` object. If the returned value is `null`, the `delete` method should return `false`. If the returned value is not `null`, `delete` should return `true` only if the attempts to remove the found reference from both `TreeSet`'s are successful. The `delete` method should return `false` otherwise.

d) `public String toStringAlphabetical()`—This method returns the `String` returned by invoking `toString()` on the `TreeSet` ordered by the natural ordering of its `Comparable` objects.

e) `public String toStringChronological()`—This method returns the `String` returned by invoking `toString()` on the `TreeSet` ordered by the `public static final Comparator` object of class `BirthInfo`.

f) `public String saveToFile(String)`—This method looks very much like the method `saveToFile` from Chapter 13. Instead of a simple `for` loop iterating through the possible array indices, you will need either an enhanced `for` loop or an `Iterator` object working with a `while` loop. You may invoke the `iterator` method on the `TreeSet` ordered by the natural ordering of its `Comparable` objects to get an `Iterator` object for that `TreeSet`, and then use a `while` loop with the `Iterator` to get a reference to each object in the `TreeSet` and write each object to the `ObjectOutputStream`.

g) `public String loadFromFile(String)`—This method will be similar to the method `loadFromFile` in the class `List<E>` class from Chapter 13. However, in this case, we do not want to invoke `loadFromFile` on each of the two `TreeSets`. If we did that, we would have two sets of the same data objects in memory that were read from the file. Instead, write `loadFromFile` in the class `BirthInfoCollection` so that each object comes just once from the file and is inserted into both `TreeSet`'s using the `add` method described above.

The Class HashSet<T> of the Java Collections API

A hash table is a data structure that does not organize its elements according to any ordering scheme. Instead it uses a hashing function to determine where items should be placed. In a Data Structures course you will study the efficiency of various hashing functions and compare hash tables to binary trees. In this text, we simply introduce the `HashSet<E>` of the Java API and present exercises that pose the problem of how to produce ordered displays of data when sorting the data structure is contrary to its basic structure and would destroy its efficiency.

`HashSet<E>` has four constructors:

1) `HashSet()`—Constructs a new, empty set; the backing `HashMap` has default capacity (16) and load factor (0.75)

2) `HashSet(Collection<? extends E> c)`—Constructs a new set containing the elements in the specified collection.

3) `HashSet(int initialCapacity)`—Constructs a new, empty set; the backing `HashMap` has the specified initial capacity and load factor (0.75)

4) `HashSet(int initialCapacity, float loadFactor)`—Constructs a new, empty set; the backing `HashMap` has the specified initial capacity and the specified load factor

The class `HashSet<E>` has a variety of methods. A complete list can be found in the Java API documentation. A selection of them that are useful to us are as follows:

- `boolean add(E e)`—Adds the specified element to this set if it is not already present.
- `boolean contains(Object o)`—Returns `true` if this set contains the specified element.
- `boolean isEmpty()`—Returns `true` if this set contains no elements.
- `Iterator<E> iterator()`—Returns an `iterator` over the elements in this set.
- `boolean remove(Object o)`—Removes the specified element from this set if it is present.

Exercise 16.28: Extension of Exercise 16.26 Revise your class named `AddressInfoCollection` so that it employs one instance variable of class `java.util.HashSet<AddressInfo>` instead of two instance variables of class `TreeSet<AddressInfo>`. Your **Database** class should need no revision. More specifically:

1) Your class `AddressInfoCollection` will have one private instance variable that is reference to an object of class `HashSet<AddressInfo>`. The `HashSet<AddressInfo>` will be constructed with the zero parameter constructor.
2) The methods of class `AddressInfoCollection` will change as described below:
   a) `public AddressInfo search(AddressInfo)`—The `search` method will invoke the contains method on the `HashSet<AddressInfo>` with argument equal to the `AddressInfo` parameter. If the `contains` method returns `false`, search should return `null`. If the `contains` method returns `true`, use an enhanced `for` loop or a `while` loop with an `Iterator<AddressInfo>` object to find the reference and return it.
   b) `public boolean add(AddressInfo)`—The `add` method will invoke the `contains` method on the `HashSet<AddressInfo>` to check for a duplicate record. If a duplicate is found, `add` should return `false`. If no duplicate is found, the `add` method will return the value returned by invoking the `add` method on the `HashSet<AddressInfo>` with argument equal to the `AddressInfo` parameter.

(continues)

Exercise 16.28, continued

c) `public boolean delete(AddressInfo)`—The `delete` method will invoke the `contains` method on the `HashSet<AddressInfo>` to see if a matching record is present in the `HashSet<AddressInfo>`. If the `contains` method returns `false`, the `delete` method should return `false`. If the `contains` method returns `true`, `delete` should return the value returned by invoking the `remove` method on the `HashSet<AddressInfo>` with argument equal to the `AddressInfo` parameter.

d) `public String toStringAlphabetical()`—Construct a local `TreeSet<AddressInfo>` with the no argument to the constructor. Invoke the `addAll` method on that `TreeSet<AddressInfo>` with argument equal to the `HashSet<AddressInfo>`. Return the `String` returned by invoking `toString` on the `TreeSet<AddressInfo>`.

e) `public String toStringByAddress()`—Construct a local `TreeSet<AddressInfo>` with the `public static final Comparator` object of class `AddressInfo` as an argument to the constructor. Invoke the `addAll` method on that `TreeSet<AddressInfo>` with argument equal to the `HashSet<AddressInfo>`. Return the `String` returned by invoking `toString` on the `TreeSet<AddressInfo>`.

f) `String saveToFile(String)`—This method looks very much like the method `saveToFile` from Chapter 13. Invoke the `iterator` method on the `HashSet<AddressInfo>` to get an `Iterator<AddressInfo>` object. Use a `while` loop with the `Iterator<AddressInfo>` to get a reference to each record in the `HashSet<AddressInfo>` and write each object to the `ObjectOutputStream`.

g) `public String loadFromFile(String)`—Write `loadFromFile` in the class `AddressInfoCollection` so that each object comes from the file and is inserted into the `HashSet` using the add method described above.

Exercise 16.29: Extension of Exercise 16.27 Revise your class `BirthInfoCollection` so that it employs one instance variable of class `java.util.HashSet<BirthInfo>` instead of two instance variables of class `TreeSet<BirthInfo>`. Your **Database** class should need no revision. More specifically:

1) Your class `BirthInfoCollection` will have one private instance variable that is reference to an object of class `HashSet<BirthInfo>`. The `HashSet<BirthInfo>` will be constructed with the zero parameter constructor.

2) The methods of class `BirthInfo Collection` will change as described below:

   a) `public BirthInfo search(BirthInfo)`—The `search` method will invoke the `contains` method on the `HashSet<BirthInfo>` with argument equal to the `BirthInfo` parameter. If the `contains` method returns `false`, search should return `null`. If the `contains` method returns `true`, use an enhanced `for` loop or a `while` loop with an `Iterator<BirthInfo>` object to find the reference and return it.

(continues)

Exercise 16.29, continued

b) `public boolean add(BirthInfo)`—The `add` method will invoke the `contains` method on the `HashSet<BirthInfo>` to check for a duplicate record. If a duplicate is found, `add` should return `false`. If no duplicate is found, the `add` method will return the value returned by invoking the `add` method on the `HashSet<BirthInfo>` with argument equal to the `BirthInfo` parameter.

c) `public boolean delete(BirthInfo)`—The `delete` method will invoke the `contains` method on the `HashSet<BirthInfo>` to see if a matching record is present in the `HashSet<BirthInfo>`. If the `contains` method returns `false`, the `delete` method should return `false`. If the `contains` method returns `true`, `delete` should return the value returned by invoking the `remove` method on the `HashSet<BirthInfo>` with argument equal to the `BirthInfo` parameter.

d) `String toStringAlphabetical()`—Construct a local `TreeSet<BirthInfo>` with the no argument to the constructor. Invoke the `addAll` method on that `TreeSet<BirthInfo>` with argument equal to the `HashSet<BirthInfo>`. Return the `String` returned by invoking `toString` on the `TreeSet<BirthInfo>`.

e) `public String toStringByAddress()`—Construct a local `TreeSet<BirthInfo>` with the `public static final Comparator` object of class `BirthInfo` as an argument to the constructor. Invoke the `addAll` method on that `TreeSet<BirthInfo>` with argument equal to the `HashSet<BirthInfo>`. Return the `String` returned by invoking `toString` on the `TreeSet<BirthInfo>`.

f) `public String saveToFile(String)`—This method looks very much like the method `saveToFile` from Chapter 13. Invoke the `iterator` method on the `HashSet<BirthInfo>` to get an `Iterator<BirthInfo>` object. Use a `while` loop with the `Iterator<BirthInfo>` to get a reference to each record in the `HashSet<BirthInfo>` and write each object to the `ObjectOutputStream`.

g) `public String loadFromFile(String)`—Write `loadFromFile` in the class `BirthInfoCollection` so that each object comes from the file and is inserted into the `HashSet` using the `add` method described above.

Lab: JavaDoc • Chapter 16

The purpose of this lab is to learn how to use the javadoc utility to generate documentation for your Java projects in HTML format. The steps of this lab assume that you are using Mac OS X. You will:

- Use javadoc to generate documentation without changing your .java files
- Read about documentation comments and special tags that you can edit into your comments to enhance your documentation
- Insert documentation comments into your code using Eclipse
- Learn how to generate javadoc documentation at the command line
- Learn how to generate javadoc documentation from within Eclipse

1) Make a copy on the **Desktop** of the folder containing your .java files for your database project. For this handout, I will use the name for_database2 as the folder's name. You should use the package name that you have already chosen for the code.
2) Launch the **Terminal** application from the dock.
3) Change directory to the **Desktop** by typing `cd Desktop` at the % prompt.
4) In order to see all of your files from the command line perspective, type `ls for_database2.`
5) Generate javadoc documentation for your code by typing `javadoc for_database2` at the % prompt.
6) Note all of the messages indicating that files are being loaded and that .html files are being generated. Also note that some files are created on the **Desktop** at the same level as your folder and some files are created inside your folder.
7) From the GUI perspective, double click the icon for the file index.html on the **Desktop**.
8) Navigate through the javadoc documentation and compare the specifications to your code. Note in particular that the private instance variables do not appear in the generated documentation. That is because the default in javadoc is to generate documentation for only the public variables, constructors, and methods.
9) Generate javadoc documentation for all your code by typing `javadoc –private for_database2` at the % prompt.
10) From the GUI perspective, double click the icon for the file index.html on the **Desktop**.
11) Navigate through the javadoc documentation and compare the specifications to your code. Note that the private instance variables do appear in the generated documentation this time.
12) Quit the browser you have been using to view the html files.
13) Within the **Terminal** application window, type `logout` at the % prompt.

14) Quit the **Terminal** application.

15) Read Chapter 19 of *The Java Programming Language, Fourth Edition* to learn about the special comments and various tags that can be inserted in your code to enhance your documentation files. You may also read about `javadoc` online by typing `man javadoc` at the `%` prompt to view the online manual entry. If you try this, you get one window's worth of reading at a time. Press the space bar to get the next window's worth. Hold down the `control` key while typing a `b` to go back a window's worth. Type a `q` to get back to the `%` prompt.

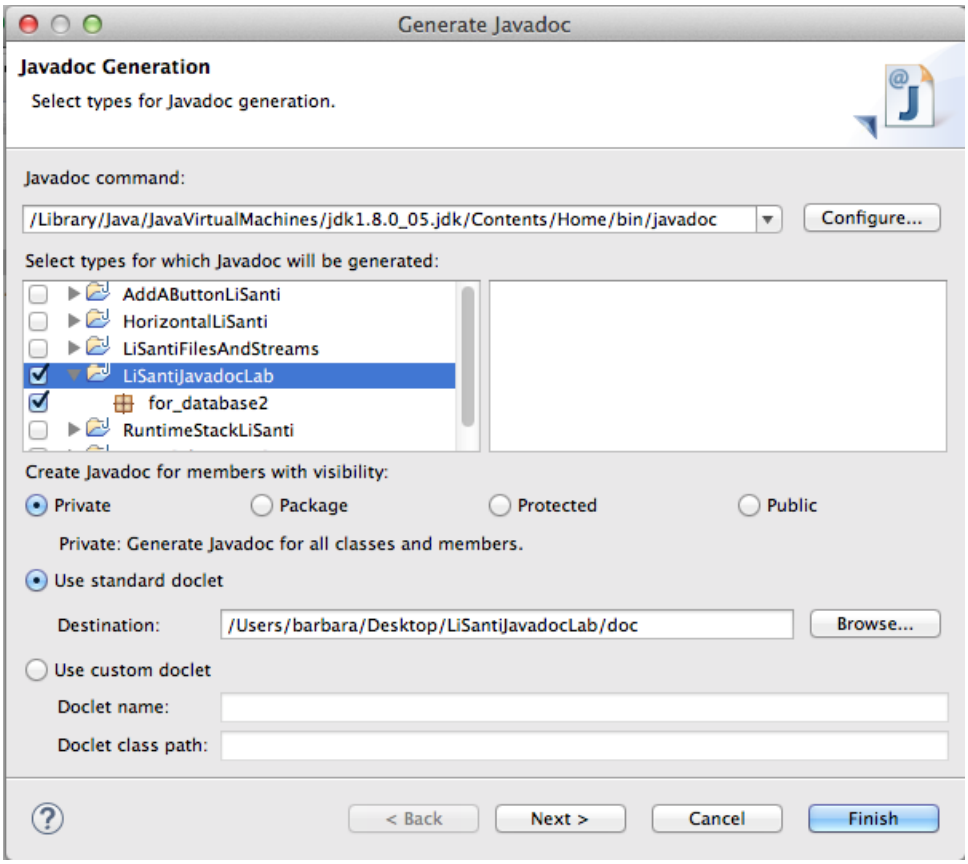
16) Build an Eclipse project named with `your name` and `JavadocLab`.

17) Add your code package named `for_database2` to the Eclipse project.

18) Insert documentation comments into your `Java` classes to document *every* class, variable, and method. Note that the Java editor in Eclipse assists you. When you type `/**` and return, the editor makes the comment light blue in color and gives a skeleton for the rest of the comment down to the closing `*/`. Use the `@author`, `@param`, and `@return` tags where appropriate. Make sure to save each of your files after editing in the documentation comments.

19) Select your code package in the `Package Explorer` view of Eclipse.

20) From the `Project` menu of Eclipse, select `Generate Javadoc`. A dialog box similar to the one below will open.

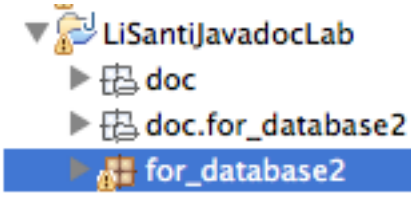


21) Make sure that your code package is selected in the left scrolling list.

22) Click the radio button labeled `Private`.

23) Note that the destination for the documentation is a folder named **doc** within your Eclipse project folder and click the **Finish** button.

24) If there are no errors, two packages named **doc** and **doc.for_database2** will appear in your project folder as shown below.



25) Click the triangle to the left of **doc** to view its contents. You should find a file named **index.html**.

26) Double click on **index.html** to start viewing the documentation within the Eclipse window. Navigate to examine all of the documentation.

27) Quit Eclipse.

28) Drag a copy of your Eclipse project folder onto a flash drive.

17

Introduction to Lambda Expressions and the Stream Abstraction of Java 8

The sources for the technical content of this chapter are two papers written in September 2013 by Brian Goetz, Oracle's Java Language Architect: State of Lambda (http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html) and State of Lambda: Libraries Edition (http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html).

INTRODUCTION TO LAMBDA EXPRESSIONS

We have made extensive use of Java's `ActionListener` interface in all of our programs that have GUI's. In order to define a class that implements the `ActionListener` interface, we must provide the details of the `actionPerformed` method of that class. The `actionPerformed` method is the only member we define within the class. Because Java is object oriented, a method cannot stand alone. It must be a member of a class. One could say that the `ActionListener` interface exists only so that we can define a class that is the home for its `actionPerformed` method. In a similar fashion the `Comparator` interface that we introduced in Chapter 12 exists so we can define its `compare` method. There are several other interfaces in Java that are implemented by providing the details of only one method. Such an interface is called a *functional interface*.

It often happens that the body of an `actionPerformed` method is very short, perhaps only one line. But the code that one must write to enclose that one line usually spans over 5 lines because there is the header for the class, the header for the `actionPerformed` method, the one line of code, and two lines for the closing curly brackets for the method and the class. Hence, to define one line of code, one writes 5 lines. Below is an example of such code taken from the class `ThreeStoogesL1` (See the code package for the lab that adds a button to the Three Stooges application at the end of Chapter 2.)

```
class CurlyBHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
            resultsTA.append("Nyuck Nyuck!\n");
    }// actionPerformed
}// CurlyBHandler
```

Assuming that curlyB has been declared to be a reference to a JButton object and that the JButton object has been declared and constructed with the following lines of code,

```
private JButton curlyB;
curlyB = new JButton("Curly");
```

the line of code for adding an ActionListener to the button curlyB is as follows.

```
curlyB.addActionListener(new CurlyBHandler());
```

Before Java 8, one could shorten the code by making the class anonymous and declaring it in the line of code that adds the ActionListener as follows:

```
curlyB.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
            resultsTA.append("Nyuck Nyuck!\n");
    }//actionPerformed
});
```

However, this was inelegant and did not improve the readability of the code.

With Java 8, when an argument to a method is a type defined by a functional interface, you can use a shortcut, called a *lambda expression*. The line below shows how to rewrite using a lambda expression.

```
curlyB.addActionListener(
    (ActionEvent e) -> resultsTA.append("Nyuck Nyuck!\n"));
```

Thinking from a functional perspective instead of an object oriented perspective, when code adds an ActionListener, it is really providing the details of a method that provides the functionality of the listener. It is the method that is important. When the Java compiler begins to parse the statement curlyB.addActionListener(, the compiler already knows that the code within the parentheses must refer to an ActionListener object and in order to be an ActionListener object, the object must provide the details of its actionPerformed method. Beginning with Java 8, the syntax of the Java language provides for *lambda expressions* that are anonymous methods and simultaneously represent an instance of the required *target type*. The general syntax of a lambda expression consists of an argument list, the arrow token formed by a dash and the greater than symbol ->, and a body for the method. The body can be either a single expression or a statement block enclosed in curly brackets. In the expression form, the body is simply evaluated and returned. In the block form, the body is evaluated like a method body. A return statement returns control to the caller of the anonymous method. If the body produces a result, every control path must return something appropriate or throw an exception.

The class that a lambda expression represents is inferred from the surrounding context. For example, if we find a lambda expression as the argument for the `addActionListener` method, it is clear that the class is one that implements the `ActionListener` interface. Hence, the lambda expression must be the anonymous form of an `actionPerformed` method.

Exercise 17.1: Revise the class `ThreeStoogesL1` from the Add a Button Lab of Chapter 2 to eliminate the classes `MoeBHandler`, `LarryBHandler`, and `CurlyBHandler` and employ lambda expressions to provide the details of each `actionPerformed` method.

Exercise 17.2: Revise the code of Sample Program 6.1, the class `DriversAide`, to eliminate the classes `RedBHandler`, `YellowBHandler`, and `GreenBHandler` and employ lambda expressions to provide the details of each `actionPerformed` method.

In the case that there is more than one line of code in the `actionPerformed` method, the use of a lambda expression might not actually improve the readability of the code. Consider the inner classes `EqualBHandler` and `ClearBHandler` of class `SumEm` of Sample Program 6.3.

```
public class EqualBHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
            String input1string, input2string;
            int input1, input2, sum;
            input1string = input1TF.getText();
            input1 = Integer.parseInt(input1string);
            input2string = input2TF.getText();
            input2 = Integer.parseInt(input2string);
            sum = input1 + input2;
            resultTF.setText(sum + "");
    } //actionPerformed
} //EqualBHandler

public class ClearBHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
            input1TF.setText("");
            input2TF.setText("");
            resultTF.setText("");
    }  //actionPerformed
} //ClearBHandler
```

Since there will be more than one line of code in the body of each lambda expression, the body is a block of code enclosed by curly brackets.

Replacing the class `ClearBHandler` with a lambda expression, we get the following lines of code:

```
clearB.addActionListener(
    (ActionEvent e) -> {
            input1TF.setText("");
            input2TF.setText("");
            resultTF.setText("");
});
```

That's not particularly difficult to read. However, replacing the class `EqualBHandler` with a lambda expression, we get the following lines of code:

```
equalB.addActionListener((ActionEvent e) -> {
    String input1string, input2string;
    int input1, input2, sum;
    input1string = input1TF.getText();
    input1 = Integer.parseInt(input1string);
    input2string = input2TF.getText();
    input2 = Integer.parseInt(input2string);
    sum = input1 + input2;
    resultTF.setText(sum + "");
});
```

Since the body of the lambda expression above invokes `parseInt` method of the Integer class, we know that a `NumberFormatException` could be thrown. The code really should employ `try` and `catch` to deal with that possibility, making it even more complicated. Since the point is to clarify the code instead of making it harder to read, the code substitution above might not be optimal. However, if there was a method was written that encapsulated the more complicated code and the body of the lambda expression simply invoked that method, then the lambda expression can remain short and straightforward. We could write a method named `handleUserInputs` within the `SumEm` class and then write an invocation of that method as the body of the lambda expression as follows:

```
private void handleUserInputs(){
    int input1, input2, sum;
    String input1string, input2string;
    try{
            input1string = input1TF.getText();
            input1 = Integer.parseInt(input1string);
            input2string = input2TF.getText();
            input2 = Integer.parseInt(input2string);
            sum = input1 + input2;
            resultTF.setText(sum + "");
    }//try
    catch(NumberFormatException nFE){
            resultTF.setText("Input was in the wrong format. Try again.");
    }//catch
}//handleUserInputs
```

```
equalB.addActionListener((ActionEvent e) -> handleUserInputs());
```

As another example, consider the class **AddColor** below that is written without lambda expressions.

```
// Three buttons will increase one each of the R, G, or B components of the
// background color of the content pane until the maximum
// is reached for each component.
// A reset button resets the background color of the content pane to black.

import java.awt.*; // for original GUI stuff
import java.awt.event.*; // for event handling
import javax.swing.*; //for swing GUI stuff
public class AddColor extends JFrame {
  private JButton redB,
      greenB,
      blueB;//user clicks these to add color to content pane's background
  private JButton resetB; // to reset all color components to 0
  private JTextField showRGBTF;//will display the RGB components of the
                               //background color of the content pane
  private Container myCP; //to hold a reference to the content pane

  public AddColor () {
    super("Increase Color Components");
    setSize(650, 300);
    setLocation(100,100);
    myCP = getContentPane();
    myCP.setLayout(null);//there is no layout manager
    myCP.setBackground(Color.BLACK);// red, green, and blue components are 0

    redB =
      makeButton(30,75,120,50, "Increase Red",Color.RED, new RedBHandler() );
    greenB =
      makeButton(180,75,120,50,"Increase Green",Color.GREEN,
                                              new GreenBHandler() );
    blueB =
      makeButton(330,75,120,50, "Increase Blue",Color.BLUE,
                                              new BlueBHandler()  );
    resetB =
      makeButton(500,75,120,50, "Reset",Color.BLACK, new ResetBHandler() );

    showRGBTF = new JTextField("R is 0, G is 0, and B is 0.");
    showRGBTF.setSize(200,30);
    showRGBTF.setLocation(225,150);
    showRGBTF.setEditable(false);
    myCP.add(showRGBTF);
```

```
        setVisible(true);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }//windowClosing

        }); //end of definition of WindowAdapter and semicolon to end the line

    } // AddColor constructor

    private JButton makeButton(int theX, int theY, int theWidth, int theHeight,
                String theText, Color theForeground, ActionListener theHandler){
        JButton toReturn = new JButton(theText);
        toReturn.setSize(theWidth, theHeight);
        toReturn.setLocation(theX, theY);
        toReturn.setForeground(theForeground);
        myCP.add(toReturn);
        toReturn.addActionListener(theHandler);
        return toReturn;
    }//makeButton

    private int handleIncrease(int theComponent, JButton theButton){
        if (theComponent + 20 > 255 ) {
                theComponent = 255;
                theButton.setEnabled(false);
        } else {
                theComponent = theComponent + 20;
        }//else
        return theComponent;
    }//handleIncrease

    private void handleOneColor(Color c){
        Color currentBackgroundColor = myCP.getBackground();
        int redComponent = currentBackgroundColor.getRed();
        int greenComponent = currentBackgroundColor.getGreen();
        int blueComponent = currentBackgroundColor.getBlue();
        if(c.equals(Color.RED)){
                redComponent = handleIncrease(redComponent, redB);
        } else if (c.equals(Color.GREEN)){
                greenComponent = handleIncrease(greenComponent, greenB);
        } else {
                blueComponent = handleIncrease(blueComponent, blueB);
        }//last else
        Color adjustedColor = new Color
                                (redComponent, greenComponent, blueComponent);
        myCP.setBackground(adjustedColor);
```

```
    showRGBTF.setText ("R is " + redComponent + ", G is " + greenComponent
                          + ", B is " + blueComponent + ".");
  }//handleOneColor

  private void reset(){
    myCP.setBackground(Color.BLACK);
    showRGBTF.setText ("R is 0, G is 0, B is 0.");
    redB.setEnabled(true);
    greenB.setEnabled(true);
    blueB.setEnabled(true);
  }//reset

  public class RedBHandler implements ActionListener {
    public void actionPerformed (ActionEvent e) {
           handleOneColor(Color.RED);
    } //actionPerformed
  } // RedBHandler

  public class GreenBHandler implements ActionListener {
    public void actionPerformed (ActionEvent e) {
           handleOneColor(Color.GREEN);
    } //actionPerformed
  } // GreenBHandler

  public class BlueBHandler implements ActionListener {
    public void actionPerformed (ActionEvent e) {
           handleOneColor(Color.BLUE);
    } //actionPerformed
  } // BlueBHandler

  public class ResetBHandler implements ActionListener {
    public void actionPerformed (ActionEvent e) {
           reset();
    } //actionPerformed
  } // ResetBHandler
  public static void main (String args[]) {
    AddColor myAppF = new AddColor ();
  } //main

} // AddColor
```

Note that the last parameter of the method `makeButton` is of type `ActionListener`. Remember that the semantics of parameter passing are the same as assignment. Hence, the corresponding argument has target type `ActionListener`, and we can replace the expressions that construct instances of `RedBHandler`, `GreenBHandler`, `BlueBHandler`, and `ResetBHandler` with lambda expressions. We eliminate the class `RedBHandler` as follows:

```
redB =
  makeButton(30,75,120,50, "Increase Red",Color.RED, new RedBHandler() );
```

becomes

```
redB = makeButton(30,75,120,50, "Increase Red",Color.RED,
                  (ActionEvent e) -> handleOneColor(Color.RED) );
```

Exercise 17.3: Write the necessary lambda expressions to replace the last argument in the other three invocations of method `makeButton` and thereby eliminate the inner classes `GreenBHandler`, `BlueBHandler`, and `ResetBHandler`.

Exercise 17.4: Revise the class `AddColor` to produce the class `AddOrSubtractColor`. `AddOrSubtractColor` should have three additional buttons that reduce the corresponding color component by 20 until the component hits the minimum of zero. When a color component is zero, the corresponding button to decrease that color component should be disabled. Note that if a button for increasing a color component has been disabled, an event that consists of clicking the corresponding decrease button should enable the increase button. Similarly, if a button for decreasing a color component has been disabled, an event that consists of clicking the corresponding increase button should enable the button. Write methods and use lambda expressions to eliminate the need for inner classes that implement the `ActionListener` interface.

In Chapter 12 we gave a preview of lambda expressions when we discussed the `Comparator<T>` interface and in particular constructed a `Comparator<Name>` `public static final` variable for the class `Name` with the following line of code.

```
public static final Comparator<Name>
    REVERSE_COMPARATOR = (name1, name2) -> -name1.compareTo(name2);
```

Since the compiler can infer that a `Comparator<Name>` must provide a `compare` method that has two `Name` parameters, our code does not need to specify the types for the parameters in the lambda expression. In this case, the lambda expression is said to be *implicitly typed*. When the lambda expression specifies the types of its parameters (as above `ActionEvent e`), the lambda expression is said to be *explicitly typed*. Hence, we could shorten the lambda expression

```
(ActionEvent e) -> handleOneColor(Color.RED)
```

to

```
(e) -> handleOneColor(Color.RED)
```

Exercise 17.5: For Exercise 12.4, nested within the class `AddressInfo`, you wrote a private static class `AddressComp` that implements the `Comparator<AddressInfo>` interface in order to construct a `public static final` variable named `ADDRESS_COMPARATOR`. Eliminate the need for class `AddressComp` by replacing the construction of an instance of class `AddressComp` with a lambda expression.

Exercise 17.6: For Exercise 12.5, nested within the class `BirthInfo`, you wrote a private static class `ChronComp` that implements the `Comparator<BirthInfo>` interface in order to construct a `public static final` variable named `DATE_COMPARATOR`. Eliminate the need for class `ChronComp` by replacing the construction of an instance of class `ChronComp` with a lambda expression.

In general lambda expressions can appear only in contexts that have target types. Those contexts are: variable declarations, assignments, return statements, array initializers, method or constructor arguments, lambda expression bodies, conditional expressions `(? : )`, and cast expressions.

Our examples above use only the `ActionListener` and `Comparator<T>` functional interfaces. There were other functional interfaces in the Java API before Java 8. In addition, Java 8 added a new package, `java.util.function`, that contains functional interfaces that are expected to be commonly used, such as:

`Predicate<T>`—a Boolean valued property of an object
`Consumer<T>`—an action to be performed on an object
`Function<T, R>`—a function transforming a `T` to an `R`
`Supplier<T>`—provide an instance of `T`(such as factory)
`UnaryOperator<T>`—a function from `T` to `T`
`BinaryOperator<T>`—a function from `(T, T)` to `T`

Method Expressions

Lambda expressions represent instances of functional interfaces and anonymous methods. It is often useful to use an already existing method. Consider the class PhoneInfo from Chapter 13:

```
public class PhoneInfo implements java.io.Serializable, Comparable<PhoneInfo> {
    private Name myName;
    private String myHomePhone;
    private String myCellPhone;
    private String myWorkPhone;
    public PhoneInfo(String theFirst, String theFamily,
                String theHomePhone, String theCellPhone, String theWorkPhone) {
            myName = new Name(theFirst,theFamily);
            myHomePhone = theHomePhone;
            myCellPhone = theCellPhone;
            myWorkPhone = theWorkPhone;
```

```
}// 5 parameter constructor

public PhoneInfo(String theFirst, String theFamily) {
        myName = new Name(theFirst,theFamily);
        myHomePhone = "none";
        myCellPhone = "none";
        myWorkPhone = "none";
}// 2 arg constructor

public Name getName() {
        return myName;
}// getName

public String getHomePhone() {
        return myHomePhone;
}// getHomePhone

public String getCellPhone() {
        return myCellPhone;
}// getCellPhone

public String getWorkPhone() {
        return myWorkPhone;
}// getWorkPhone

public void setName(Name theName) {
        myName = theName;
}// setFamily

public void setHomePhone(String theHomePhone) {
        myHomePhone = theHomePhone;
}// setHomePhone

public void setCellPhone(String theCellPhone) {
        myCellPhone = theCellPhone;
}// setCellPhone

public void setWorkPhone(String theWorkPhone) {
        myWorkPhone = theWorkPhone;
}// setWorkPhone

public String toString() {
        return "Name: " + myName + "\n"
                +  "Home Phone Number: " + myHomePhone + "\n"
                +  "Cell Phone Number: " + myCellPhone + "\n"
                +  "Work Phone Number: " + myWorkPhone + "\n";
}// toString

public int compareTo(PhoneInfo theOther){
                return myName.compareTo(theOther.myName);
```

```
    }// compareTo

    public boolean equals (Object compareObj) {
            PhoneInfo thePI = (PhoneInfo)compareObj;
            return (myName.equals(thePI.myName));
    }// equals

}// PhoneInfo
```

The class `PhoneInfo` implements the `Comparable<PhoneInfo>` interface through its `compareTo` method. We might want to construct lists of `PhoneInfo` objects that are each ordered by one of the three kinds of phone numbers. We have `get` methods for each phone number field. Those fields are of class `String`. `String`'s are `Comparable` because the `String` class has its own `compareTo` method.

Java 8 added the ability to place `static` methods in functional interfaces. You can think of those `static` methods as helper methods that are specific to the interface and are allowed to live within the interface rather than in a separate class. Java 8 supplied several versions of a `static comparing` method to the `Comparator<T>` interface. One of those comparing methods takes as an argument a method that extracts a `Comparable` sort key from type `T` and returns a `Comparator<T>` that compares by that sort key. In other words, if we supply a `get` method to the `static comparing` method of the `Comparator<T>` interface, it will return a `Comparator<T>` object whose `compare` method is based on that `get` method. The new idea here is that the reference we pass to the `comparing` method is a reference to a method, not a reference to an object. To illustrate, consider the following code that results in three `Comparator<PhoneInfo>` objects being created:

```
Comparator<PhoneInfo> byWork = Comparator.comparing(PhoneInfo::getWorkPhone);
Comparator<PhoneInfo> byCell = Comparator.comparing(PhoneInfo::getCellPhone);
Comparator<PhoneInfo> byHome = Comparator.comparing(PhoneInfo::getHomePhone);
```

The new syntax here is the double colon operator `::` Its meaning is clear. Each `get` method is extracted from `class PhoneInfo` and sent to the `static comparing` method of the `Comparator` interface. The `comparing` method returns a `Comparator<PhoneInfo>` instance.

There are several different kinds of method references, each with slightly different syntax:

- A `static` method (`ClassName::methodName`)
- An instance method of a particular object (`instanceReference::methodName`)
- A super method of a particular object (`super::methodName`)
- An instance method of an arbitrary object of a particular type (`ClassName::methodName`)
- A class constructor reference (`ClassName::new`)
- An array constructor reference (`TypeName[]::new`)

The following sample code shows the use of the method references to create `Comparator` objects and the subsequent construction of `Lists` whose ordering is based on the `Comparator` object supplied to the `List` constructor. We use the `List` class of Chapter 12.

```
import java.util.*;
public class TestMethodReferences {
    public static void main(String[] args) {
    PhoneInfo p1, p2, p3;
    p1 = new PhoneInfo("Barbara", "Li Santi", "510-345-6789",
                              "510-456-7890", "510-430-2247");
    p2 = new PhoneInfo("Lydia", "Mann", "510-567-8901",
                              "510-678-9012","415-123-4567");
    p3 = new PhoneInfo("George", "Mann", "415-987-6543",
                              "415-876-5432", "415-234-5678");
    Comparator<PhoneInfo> byWork =
                        Comparator.comparing(PhoneInfo::getWorkPhone);
    Comparator<PhoneInfo> byCell =
                        Comparator.comparing(PhoneInfo::getCellPhone);
    Comparator<PhoneInfo> byHome =
                        Comparator.comparing(PhoneInfo::getHomePhone);

    List<PhoneInfo> workList = new List<> (byWork);
    List<PhoneInfo> cellList = new List<> (byCell);
    List<PhoneInfo> homeList = new List<> (byHome);

    if(workList.add(p1) &&workList.add(p2)&& workList.add(p3)){
            System.out.println("Original work list\n" + workList);
            workList.bubbleSort ();
            System.out.println("work list after sorting\n" + workList);
    }//workList

    if(cellList.add(p1) &&cellList.add(p2)&& cellList.add(p3)){
            System.out.println("Original cell list\n" + cellList);
            cellList.bubbleSort ();
            System.out.println("cell list after sorting\n" + cellList);
    }//cellList

    if(homeList.add(p1) &&homeList.add(p2)&& homeList.add(p3)){
            System.out.println("Original home list\n" + homeList);
            homeList.bubbleSort ();
            System.out.println("home list after sorting\n" + homeList);
    }//homeList
  }//main
}//TestMethodReferences
```

INTRODUCTION TO THE STREAM ABSTRACTION

In all of the code we have written so far, whenever we examine each item in a data structure we do it sequentially. It is as though we line up the data items and process them one at a time. Modern computers have the capacity for parallel processing. In the parallel processing model, the items are divided into groups, the groups are processed simultaneously and the results are combined to produce the result for the aggregate of data items.

Both the traditional `for` loop that we used extensively in conjunction with arrays and the enhanced `for` loop introduced in Chapter 16 are examples of sequential processing. The `for` loop index of the traditional `for` loop clearly accesses the elements in an array sequentially. It may be a bit more hidden, but the enhanced `for` loop relies on an `Iterator<E>` object to present the elements in a collection sequentially. These `for` loops are an example of *external iteration*, meaning that the control of the enumeration of the elements of the data structure comes from outside the data structure itself. Hence, there was nothing in the API to provide the opportunity to manage the flow of control, which might be able to provide better performance by exploiting reordering of the data, parallelism, short-circuiting, or laziness (not processing elements that do not need to be processed).

The `Stream<T>` interface was added to the Java 8 API to address this issue. The `Stream<T>` interface supports internal iteration where the programmer's code delegates to the library code in the API the job of how best to process the aggregate of data items in the collection. The `Stream<T>` interface is used in conjunction with the functional interfaces in the package `java.util.function` that were mentioned earlier in this chapter. Hence, use of the `Stream<T>` interface goes hand in hand with lambda expressions. The idea here is not to write classes that implement the `Stream<T>` interface. There are 32 abstract methods in the `Stream<T>` interface. The point is that the Collections API that has been widely used since Java 1.5 has been enhanced to use the `Stream<T>` interface. The `Collection<E>` interface now has a `stream()` method whose return type is `Stream<E>`. Hence, we can produce a `Stream<E>` from any of the concrete classes that implement the `Collection<E>` interface. Two of those classes introduced in Chapter 16 are `TreeSet<E>` and `HashSet<E>`.

Let's look at an example of how this all fits together. The `Predicate<T>` interface is a functional interface that has as its one abstract method:

```
boolean  test(T t)
```

that evaluates this predicate on the given argument.

The `Stream<T>` interface has a method

```
boolean anyMatch(Predicate<? super T> predicate)
```

that returns whether any elements of this stream match the provided predicate.

The `Stream<T>` interface has a method

```
Stream<T> filter(Predicate<? super T> predicate)
```

that returns a stream consisting of the elements of this stream that match the given predicate.

The `Stream<T>` interface has a method

```
Optional<T> findFirst()
```

that returns an `Optional` describing the first element of this stream, or an empty `Optional` if the stream is empty.

The `Optional<T>` class has a method

```
T get()
```

If a value is present in this `Optional`, returns the value, otherwise throws `NoSuchElementException`.

With all of these pieces in hand, we can rewrite the search method that we wrote at the end of Chapter 16 that will work with a `TreeSet<PhoneInfo>` using the stream abstraction as follows:

```
public PhoneInfo search(PhoneInfo findMe){
    if(!alphaTreeSet.stream()
            .anyMatch(p -> p.equals(findMe))){
            return null;
    }//no matching object in data structure
    return alphaTreeSet.stream()
                    .filter(p -> p.equals(toSearch))
                    .findFirst()
                    .get();
}//search
```

Exercise 17.7: For Exercise 16.26, you revised the class `AddressInfoCollection` so that it employed two `TreeSet<AddressInfo>` objects. Revise the search method of class `AddressInfoCollection` again so that it uses the stream abstraction.

Exercise 17.8: For Exercise 16.27, you revised the class `BirthInfoCollection` so that it employed two `TreeSet<BirthInfo>` objects. Revise the search method of class `BirthInfoCollection` again so that it uses the stream abstraction.

Exercise 17.9: For Exercise 16.28, you revised the class `AddressInfoCollection` so that it employed a single `HashSet<AddressInfo>` object. Revise the search method of class `AddressInfoCollection` again so that it uses the stream abstraction.

Exercise 17.10: For Exercise 16.29, you revised the class `BirthInfoCollection` so that it employed a single `TreeSet<BirthInfo>` object. Revise the search method of class `BirthInfoCollection` again so that it uses the stream abstraction.