chapter 3

ADTs Unsorted List and Sorted List

Goals

Measurable goals for this chapter include that you should be able to

- describe the List ADT at a logical level
- classify list operations into the categories constructor, iterator, observer, and transformer
- identify the pre- and postconditions of a given list operation
- use the list operations to implement utility routines such as the following application-level tasks:
  - Print the list of elements
  - Create a list of elements from a file of element information
  - Store a list of elements on a file
- implement the following list operations for both unsorted lists and sorted lists:
  - Create a list
  - Determine whether the list is full
  - Determine the size of the list
  - Insert an element
  - Retrieve an element
  - Delete an element
  - Reset the list and repeatedly return the next item from the list
- explain the use of Big-O notation to describe the amount of work done by an algorithm
- compare the unsorted list operations and the sorted list operations in terms of Big-O approximations
- describe uses of Java's *abstract class* and *interface* constructs with respect to defining ADTs
- design and create classes for use with a generic list
- use a List ADT as a component of a solution to an application problem

This chapter centers on the List ADT: its definition, its implementation, and its use in problem solving. In addition to learning about this important data structure, this material should help you understand the relationships among the logical, application, and implementation levels of an ADT. In the course of the exploration of these topics, several Java constructs for supporting abstraction are introduced. Seeing how these constructs are used should enhance your appreciation for the power of abstraction. We also introduce in this chapter an analysis tool, Big-O notation, which allows us to compare the efficiency of different ADT implementations.

3.1 Lists

We all know intuitively what a list is; in our everyday lives we use lists all the time—grocery lists, lists of things to do, lists of addresses, lists of party guests.

In computer programs, lists are very useful abstract data types. They are members of a general category of abstract data types called containers; containers hold other objects. There are languages in which the list is a built-in structure. In Lisp, for example, the list is the main data type provided in the language. Although list classes are provided in the Java Class Library, the techniques for building lists and other abstract data types are so important that we show you how to design and write your own.

Linear relationship Each element except the first has a unique predecessor, and each element except the last has a unique successor

Length The number of items in a list; the length can vary over time

Unsorted list A list in which data items are placed in no particular order; the only relationship between data elements is the list predecessor and successor relationships

Sorted list A list that is sorted by the value in the key; there is a semantic relationship among the keys of the items in the list

From a programming point of view, a list is a homogeneous collection of elements, with a linear relationship between its elements. A linear relationship means that, at the logical level, each element on the list except the first one has a unique predecessor and each element except the last one has a unique successor. (At the implementation level, there is also a relationship between the elements, but the physical relationship may not be the same as the logical one.) The number of items on the list, which we call the length of the list, is a property of a list. That is, every list has a length.

Lists can be unsorted—their elements may be placed into the list in no particular order—or they can be sorted. For instance, a list of numbers can be sorted by value, a list of strings can be sorted alphabetically, and a list of grades can be sorted numerically.

When the elements in a sorted list are of composite types, we can define their logical order in many different ways. For example, suppose we have a list of student information, with each student represented by their first name, last name, identification number, and three test scores. Some of the ways we can sort such a list are:

- by last name, alphabetically
- by last name, alphabetically, and then by first name, alphabetically (in other words, the first name is used to determine relative ordering if two or more last names are identical)
- by identification number
- by average test score

If the sort order is determined directly by using the student information, such as in the first three approaches, we say that that information represents the key for the list element. In the first approach, the <last name> is the key; in the second approach, the combination of <last name – first name> is the key; and in the third approach, the <identification number> is the key. If a list cannot contain items with duplicate keys, it is said to have a unique key. In this example, the best candidate for use as a unique key is the identification number, since it is likely to have a unique value for each student in a school.

Key The attributes that are used to determine the logical order of the items on a list

This chapter deals with many kinds of lists. We make the assumption that our lists are composed of unique elements. We point out the ramifications of dropping this assumption on our list abstractions and implementations at various places within the chapter. When sorted, our lists are sorted from smallest to largest key value, though it is certainly possible to sort them largest to smallest should your application need this.

There are two basic approaches to implementing container structures such as lists: the "by copy" approach and the "by reference" approach. For our lists in this chapter, we use the "by copy" approach. This means that when a client program inserts an item into our lists, it is actually a copy of the item that is placed on the list. In addition, when an item is retrieved from our list by a client program, it is a copy of the item on the list that is returned to the program. We use the alternate approach, storing and returning references to the items instead of copies of the items, for other container structures starting in Chapter 4. At that point we discuss more thoroughly the important differences between the two approaches.

Progressing through the chapter, we develop unsorted and sorted lists of strings, sorted lists of generic elements, and in the case study, a sorted list of house information for a real estate application. As we progress, we introduce both the Java *abstract class* mechanism and the Java *interface* mechanism to help refine our list ADTs and make them more generally usable. Each time we implement a new form of list, we include the corresponding UML diagram. Each figure that displays a UML diagram includes all of the previous diagrams, so that you easily can compare the implementation approaches.

3.2 Abstract Data Type Unsorted List

Logical Level

There are many different operations that programmers can provide for lists. For different applications we can imagine all kinds of things users might need to do to a list of elements. In this chapter we formally define a list and develop a set of general-purpose operations for creating and manipulating lists. By doing this, we are building an abstract data type.

To create the definition of a list as an abstract data type, we must identify a set of operations that allow us to access and manipulate the list. In this section we design the specifications for a List ADT where the items on the list are unsorted; that is, there is no semantic relationship between an item and its predecessor or successor. They simply appear next to one another on the list.

Abstract Data Type Operations

Designing an ADT to be used by many applications is not the same as designing an application program to solve a specific problem. In the latter case we can use CRC cards to enact scenarios of the application's use, allowing us to identify and fix holes in our design before turning to implementation. Identifying scenarios for use of a general ADT is not as straightforward. We must stand back and consider what operations every user of the data type would want it to provide.

Recall that there are four categories of operations: constructors, transformers, observers, and iterators. We begin by reviewing each category and considering which List ADT operations fit into the respective categories.

Constructors A constructor creates a new instance of the data type. In Java, it is a public method with the same name as the ADT's class name. There is one piece of information that our ADT needs from the client to construct an instance of the list data type: the maximum number of items to be on the list. As this information varies from application to application, it is logical for the client to have to provide it. We can also define a default list size to be used in case the client does not provide the information.

At the end of the previous chapter we suggested that it is a good idea to include a copy constructor when defining an ADT. A *copy constructor* accepts an instance of the ADT as a parameter and creates a copy of it. Copy constructors are most appropriate when the ADT implements an unstructured composite type, such as the `Date` and `Circle` examples of the previous chapters. Although there can be situations in which a copy constructor can be helpful for an application programmer who is using a structured composite type such as a list, these situations are rare. We do not define a copy constructor for our List ADTs.

Transformers Transformers are operations that change the content of the structure in some way. A common transformer is one that makes the structure empty. However, in Java, the constructor methods associate a new, empty structure with the current instance of the ADT, effectively making it empty. Therefore, we do not need another method for making the list empty. We do need transformers to put an item into the structure, or to remove a specific item from the structure. For our Unsorted List ADT, let's call these transformers `insert` and `delete`.

Note that, since we implement our operations as object methods, the list is the object through which the method is invoked, and therefore the list itself is available to the method for manipulation. The `insert` and `delete` methods need an additional parameter: the item to be inserted or deleted. For this Unsorted List ADT, let's assume that the item to be inserted is not currently on the list and the item to be deleted is on the list.

A transformer that takes two sorted lists and merges them into one sorted list or appends one list to another is a *binary transformer*. The specification for such an operation is given in the exercises, where you are asked to implement it.

Observers Observers also come in several forms. They ask true/false questions[1] about the ADT (Is the structure empty?). They select or access a particular item (Give me a copy of the last item.). Or they return a property of the structure (How many items are in the structure?). The Unsorted List ADT needs at least two observers: `isFull` and `lengthIs`. The `isFull` observer method returns `true` if the list is full, `false` otherwise; `lengthIs` tells us how many items are on the list, as opposed to the maximum capacity of the list.

If an abstract data type places limits on the component type, we could define other observers. For example, if we know that our abstract data type is a list of numerical values, we could define statistical observers such as `minimum`, `maximum`, and `average`. Here, at the logical level, we are interested in generality; we know nothing about the type of the items on the list, so we use only general observers.

If we make the client responsible for checking for error conditions, we must make sure that the ADT gives the user the tools with which to check for the conditions. The operations that allow the client to determine whether an error condition occurs are observers. Since we are assuming that our list does not include duplicate elements, we should provide an observer that searches the list for an item with a particular key and returns whether or not the item has been found. Let's call this one `isThere`. The application programmer can use the `isThere` observer to prevent insertion of a duplicate item into the list. For example:

```
if (!list.isThere(item)) list.insert(item);
```

Iterators Iterators are used with composite types to allow the user to process an entire structure, component by component. To give the user access to each item in sequence, we provide two operations: one to initialize the iteration process (analogous to Reset or Open with a file) and one to return a copy of the "next component" each time it is called. The user can then set up a loop that processes each component. Let's call these operations `reset` and `getNextItem`. Note that `reset` is not an iterator, but is an auxiliary operation that supports the iteration. Another type of iterator is one that takes an operation and applies it to every element on the list.

Element Types Before we can formalize the specification for the Unsorted List ADT, we must consider the type of items to be held on the list. Later in the chapter we

[1]A method that returns a boolean value defined on a set of objects is sometimes called a *predicate*, with the term *observer* used for methods that inquire about an instance variable of an object.

learn how to define a generic list—a list that can hold elements of many different types. For now, so that we can concentrate on the definition and implementation of the list operations, we limit ourselves to working with a list of strings. Therefore, we call our ADT `UnsortedStringList`. In order to keep our analysis as generally applicable as possible, we still refer to list components as "elements" or "items," rather than as "strings," and we call our ADT the Unsorted List ADT in much of our discussion.

Unsorted List ADT Specification

Structure:

The list elements are Strings. The list contains unique elements; i.e., no duplicate elements as defined by the key of the list. The list has a special property called the current position—the position of the next element to be accessed by `getNextItem` during an iteration through the list. Only `reset` and `getNextItem` affect the current position.

Definitions (provided by user):

maxItems:	An integer specifying the maximum number of items to be on this list.

Operations (provided by Unsorted List ADT):

void UnsortedStringList (int maxItems)

Effect:	Instantiates this list with capacity of `maxItems` and initializes this list to empty state.
Precondition:	`maxItems` > 0
Postcondition:	This list is empty.

void UnsortedStringList ()

Effect:	Instantiates this list with capacity of 100 and initializes this list to empty state.
Postcondition:	This list is empty.

boolean isFull ()

Effect:	Determines whether this list is full.
Postcondition:	Return value = (this list is full)

int lengthIs ()

Effect:	Determines the number of elements on this list.
Postcondition:	Return value = number of elements on this list

boolean isThere (String item)

Effect:	Determines `item` is on this list.
Postcondition:	Return value = (`item` is on this list)

void insert (String item)

Effect:	Adds copy of `item` to this list.
Preconditions:	This list is not full.
	`item` is not on this list.
Postcondition:	`item` is on this list.

void delete (String item)

Effect:	Deletes the element of this list whose key matches `item` 's key.
Precondition:	One and only one element on this list has a key matching `item` 's key.
Postcondition:	No element on this list has a key matching the argument `item` 's key.

void reset ()

Effect:	Initializes current position for an iteration through this list.
Postcondition:	Current position is first element on this list.

String getNextItem ()

Effect:	Returns a copy of the element at the current position on this list and advances the value of the current position.
Preconditions:	Current position is defined.
	There exists a list element at current position.
	No list transformers have been called since most recent call to `reset`.
Postconditions:	Return value = (a copy of element at current position)
	If current position is the last element then current position is set to the beginning of this list; otherwise, it is updated to the next position.

In this specification, the responsibility of checking for error conditions is put on the user through the use of preconditions that prohibit the operation's call if these conditions exist. Recall that we call this approach programming "by contract." We have given the user the tools, such as the `isThere` operation, with which to check for the conditions. Another alternative would be to define an error variable, have each operation record whether an error occurs, and provide operations that test this variable. A third alternative would be to let the operations detect error conditions and throw appropriate exceptions. We use programming by contract in this chapter so that we can concentrate on the list abstraction and the Java constructs that support it, without having to address the extra complexity of formally protecting the operations from misuse. We use other error-handling techniques in later chapters.

The specification of the list is somewhat arbitrary. For instance, the overall assumption about the uniqueness of list items could be dropped. This is a design choice. If we were designing a specification for a specific application, then the design choice would be based on the requirements of the problem. We made an arbitrary decision not to allow duplicates. Allowing duplicates in this ADT implies changes in several operations. For example, instead of deleting an element based on its value, we might require a method that deletes an element based on its position on the list. This, in turn, might require a method that returns the position of an item on the list based on its key value.

Additionally, assumptions about specific operations could be changed—for example, we specified in the preconditions of `delete` that the element to be deleted must exist on the list. It would be just as legitimate to specify a delete operation that does not require the element to be on the list and leaves the list unchanged if the item is not there. Perhaps that version of the `delete` operation would return a `boolean` value, indicating whether or not an element had been deleted. We could even design a list ADT that provided both kinds of delete operations. In the exercises you are asked to explore and make some of these changes to the List ADTs.

Application Level

The set of operations that we are providing for the Unsorted List ADT may seem rather small and primitive. However, this set of operations gives you the tools to create other special-purpose routines that require knowledge of what the items on the list represent. For instance, we have not included a print operation. Why? We don't include it because in order to write a good print routine, we must know what the data members represent. The application programmer (who does know what the data members look like) can use the `lengthIs`, `reset`, and `getNextItem` operations to iterate through the list, printing each data member in a form that makes sense within the application. In the code that follows, we assume the desired form is a simple numbered list of the string values. We have emphasized the lines that use the list operations.

```
void printList(PrintWriter outFile, UnsortedStringList list)
// Effect: Prints contents of list to outFile
// Pre:    List has been instantiated
//         outFile is open for writing
// Post:   Each component in list has been written to outFile
//         outFile is still open
{
  int length;
  String item;
  list.reset();
  length = list.lengthIs();
  for (int counter = 1; counter <= length; counter++)
  {
    item = list.getNextItem();
    outFile.println(counter + ". " + item);
  }
}
```

For example, if the list contains the strings "`Anna Jane`", "`Joseph`", and "`Elizabeth`", then the output would be:

1. Anna Jane
2. Joseph
3. Elizabeth

Note that we defined a local variable `length`, stored the result of `list.lengthIs()` in it, and used the local variable in the loop. We could have just used the method call directly in the loop:

```
for (int counter = 1; counter <= list.lengthIs(); counter++)
```

We used the other approach for efficiency reasons. That way the `lengthIs` method is called only once, saving the overhead of extra method calls.

In the `printList` method, we made calls to the list operations specified for the Unsorted List ADT, printing a list without knowing how the list is implemented. At an application level, the operations we used (`reset`, `lengthIs`, and `getNextItem`) are logical operations on a list. At a lower level, these operations are implemented by Java methods, which manipulate an array or other data-storing medium that holds the list's elements. There are many functionally correct ways to implement an abstract data type. Between the user picture and the eventual representation in the computer's memory, there are intermediate levels of abstraction and design decisions. For instance, how is the logical order of the list elements reflected in their physical ordering? We address questions like this as we now turn to the implementation level of our ADT.

Implementation Level

There are two standard ways to implement a list. We look at a *sequential array-based list implementation* in this chapter. The distinguishing feature of this implementation is that the elements are stored sequentially, in adjacent slots in an array. The order of the elements is implicit in their placement in the array.

The second approach, which we introduce in Chapter 5, is a *linked-list implementation*. In a linked implementation, the data elements are not constrained to be stored in physically contiguous, sequential order; rather, the individual elements are stored "somewhere in memory," and their order is maintained by explicit links between them.

Before we go on, let's establish a design terminology for our list algorithms that's independent of the implementation and type of items stored on the list. Doing this allows us to describe algorithms that are valid no matter which of the standard approaches we use.

List Design Terminology Assuming that location "accesses" a particular list element,

location.node( )	Refers to all the data at location, including implementation-specific data.
location.info( )	Refers to the application data at location.
last.info( )	Refers to the application data at the last location on the list.
location.next( )	Gives the location of the node following location.node( ). If location is the end of the list, it gives the first location of the list.

A few clarifications are needed. What is meant by "all the data" at a location, and "the application data" at a location? Remember that although we are currently dealing with lists of strings, we eventually expand the kinds of elements we can use to any kind of data. The "application data" refers to the data from the application associated with a list element. In addition to the application data, a list element might have certain information associated with it, related to the implementation of the list; for example, a variable holding the location of the next list element. By "all the data" we mean the application data plus the implementation data, if there is any.

What then is location? For an array-based implementation, location is an index, because we access array slots through their indexes. For example, the design statement

```
Print location.info()
```

means "Print the application data in the array slot at index location;" eventually it might be coded in Java within the array-based implementation as

```
outFile.println(list.info[location]);
```

When we look at the linked implementation in Chapter 5, the code implementing the design statement is quite different, but the design statement itself remains the same. Thus, using this design notation, we define implementation-independent algorithms for our Unsorted List ADT. Hopefully, we can design our list algorithms just once using the design notation and then implement them using either of the implementation approaches.

What does location.next( ) mean in an array-based sequential implementation? To answer this question, consider how we access the next list element stored in an array: We increment the location index. The design statement

```
Set location to location.next()
```

might be coded in Java within the array-based implementation as

```
if (location == numItems - 1)       // Location is an array index
  location = 0;
else
  location++;
```

We have not introduced this list design terminology just to force you to learn the syntax of another computer language. We simply want to encourage you to think of the list, and the parts of the list elements, as *abstractions*. At the design stage, the implementation details can be hidden. There is a lower level of detail that is encapsulated in the "methods" node, info, and next. Using this design terminology, we hope to record algorithms that can be coded for both array-based and linked implementations.

Instance Variables

In our implementation, the elements of a list are stored in an array of String objects.

```
String[] list;
```

There are two size-related attributes of the list: capacity and current length. The *capacity* of the list is the maximum number of elements that can be stored on the list. We do not need an instance variable to hold the capacity of the list since we can use the array attribute length to determine the capacity of the list at any point within our implementation. In other words, the capacity of our list is the length of the underlying array: list.length.

However, we do need an instance variable to keep track of the current number of items we have stored in the array (also known as the current length of the list). We name this variable numItems. This variable can also be used to record where the last item was stored. Because the list items are unsorted, when we put the first item into the list, we place it into the first slot; the second item goes in the second slot, and so forth. Because our language is Java, we must remember that the first slot is indexed by 0, the second slot by 1, and the last slot by list.length - 1. Now we know where the list begins—in the first array slot. Where does the list end? The array ends at the slot with index list.length - 1, but the list ends in the slot with index numItems - 1. For example, if the list currently holds 5 items, they are kept in array locations 0 through 4; the value of the numItems instance variable is 5; and the next array location to insert a new item is also 5.

Is there any other information about the list that we must include? Both operations reset and getNextItem refer to a "current position." What is this current position? It is the index of the last element accessed in an iteration through the list. We need an instance variable to keep track of the current position. Let's call it currentPos. The method reset initializes currentPos to 0. The method getNextItem returns the value in list[currentPos] and increments currentPos. The ADT specification states that only reset and getNextItem affect the current position. Figure 3.1 illustrates the instance variables of our class UnsortedStringList. Here is the beginning of the class

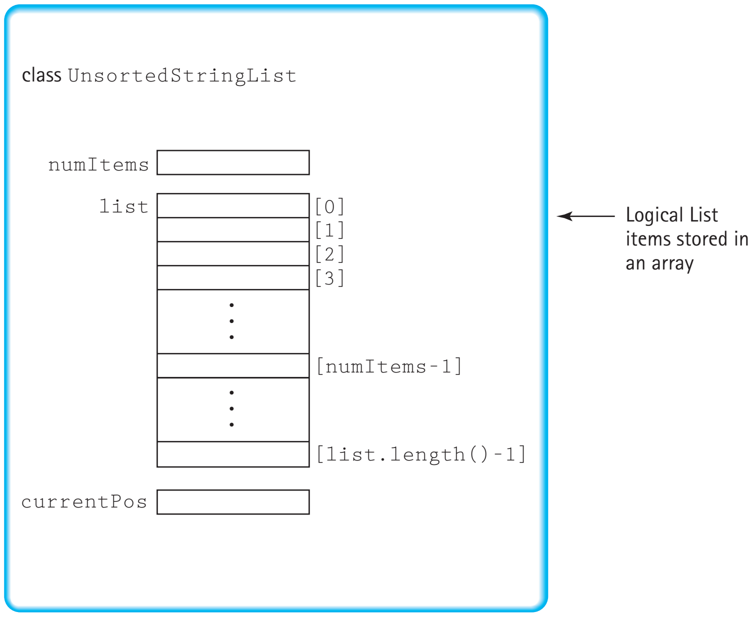


Figure 3.1 *Instance variables of Unsorted List ADT*

file, which includes the variable declarations. Note that it also includes an introductory comment and a *package* statement. The `UnsortedStringList` class is the first of several string list classes we develop. We collect all these classes together into a single package called `ch03.stringLists` (the class files can be found in the subdirectory `stringLists` or in the subdirectory `ch03` of the directory `bookFiles` on our web site).

```
//----------------------------------------------------------------------------
// UnsortedStringList.java         by Dale/Joyce/Weems              Chapter 3
//
// Defines all constructs for an array-based list of strings that is not
// kept sorted
//----------------------------------------------------------------------------

package ch03.stringLists;

public class UnsortedStringList
{
  protected String[] list;    // Array to hold list elements
  protected int numItems;     // Number of elements on this list
  protected int currentPos;   // Current position for iteration
⋮
```

Notice that we use the `protected` visibility modifier for each of the variables. Recall that this means that the variables can be "seen" only by the methods of the `UnsortedStringList` class or its subclasses. We use this approach because we create a subclass later in this chapter that needs access to the variables. This type of visibility still protects the variables from direct access by the applications that use the class.

A design choice we wish to point out, but choose not to use, is to write an `ArrayList`-based class for use here. Since the `ArrayList` class provides a variable-sized array, we could allow the underlying implementation to shrink and grow to mirror the changes in the size of the list. We would not have to deal with a "max items" constraint, so we would not need to list preconditions such as "list is not full." You are asked to investigate this alternative in the exercises.

Constructors

Now let's look at the operations that we have specified for the Unsorted List ADT. The first two operations are constructors that create empty lists. Remember that a class constructor is a method having the same name as the class, but having no return type. A constructor's purpose is to instantiate an object of the class, to initialize variables and, if necessary, to allocate resources (usually memory) for the object being constructed. Like any other method, a constructor has access to all variables and methods of the class. A new list is created empty; that is, the number of items is 0.

Our first constructor requires a positive integer parameter, which indicates the size for the underlying array.

```
public UnsortedStringList(int maxItems)
// Instantiates and returns a reference to an empty list object with
// room for maxItems elements
{
  numItems = 0;
  list = new String[maxItems];
}
```

The code for this constructor is straightforward and requires no further explanation. We have decided not to include a restatement of the method preconditions and postconditions, established in the ADT specification, when listing our code. In some cases we provide multiple versions of the same method, and we believe repeated listing of these conditions is redundant and would make for tedious reading. Therefore, we list these conditions only when we define the logical-level view of our ADTs. Nevertheless, we encourage you to always include preconditions and postconditions in comments at the beginning of your methods. Code that is meant to be used needs such documentation, but in this text, where we're already explaining the code in great detail, the comments aren't as necessary.

Our second constructor does not have a parameter. In this case, the default size of the underlying array is 100.

```
public UnsortedStringList()
// Instantiates and returns a reference to an empty list object
// with room for 100 elements
{
  numItems = 0;
  list = new String[100];
}
```

Notice that these two methods have the same name: `UnsortedStringList`. How is this possible? Remember that in the case of methods, Java uses more than just the name to identify them; it also uses the parameter list. A method's name, the number and type of parameters that are passed to it, and the arrangement of the different parameter types within the list, combine into what Java calls the signature of the method.

Signature The distinguishing features of a method heading. The combination of a method name with the number and type(s) of its parameters in their given order

Overloading The repeated use of a method name with a different signature

Java allows us to use the name of a method as many times as we wish, as long as each one has a different signature. When we use a method name more than once, we are overloading its identifier. The Java compiler needs to be able to look at a method call and determine which version of the method to invoke. The two constructors in class `UnsortedStringList` both have different signatures: One takes no arguments, the other takes an `int`. Java decides which version to call according to the arguments in the statement that invokes `UnsortedStringList`.

Simple Observers

The first nonconstructor operation, `isFull`, just checks to see whether the current number of items on the list is equal to the length of the array.

```
public boolean isFull()
// Returns whether this list is full
{
  return (list.length == numItems);
}
```

The body of the observer object method `lengthIs` is also just one statement.

```
public int lengthIs()
// Returns the number of elements on this list
{
  return numItems;
}
```

So far, we have not used our special design terminology. The algorithms have all been straightforward. The next operation, `isThere`, is more complex.

isThere Operation

The `isThere` operation allows the application programmer to determine whether a list item with a specified key exists on the list. In the case of the string list, the key is simply the string value. This string value is input to the method in the parameter `item`. A `boolean` value is returned by the method—if the string `item` matches a string on the list, `true` is returned; otherwise, `false` is returned.

Because the list items are unsorted, we must use a linear search. We begin at the first component on the list and loop until either we find a component equal to the parameter or there are no more strings to examine. Recall from Chapter 2 that we have two ways to see if two strings are the same; we could use the `equals` method of the `String` class or the `compareTo` method of the `String` class. We choose to use the `compareTo` method, since we also use it in other parts of the list implementation. Recall that this method returns a 0 if the strings are equal. Therefore, we can code

```
if (item.compareTo(list[location]) == 0)
  found = true;
```

But how do we know when to stop searching if we do not find the string? If we have examined the last element of the list, we can stop. Thus, in our design terminology, we keep looking as long as we have not examined last.info( ).We summarize these observations in the algorithm below.

isThere (item): returns boolean

```
Initialize location to position of first list element
Set found to false
Set moreToSearch to (have not examined last.info())

while moreToSearch AND NOT found
  if item.compareTo(location.info()) == 0
    Set found to true
  else
    Set location to location.next()
    Set moreToSearch to (have not examined last.info())

return found
```

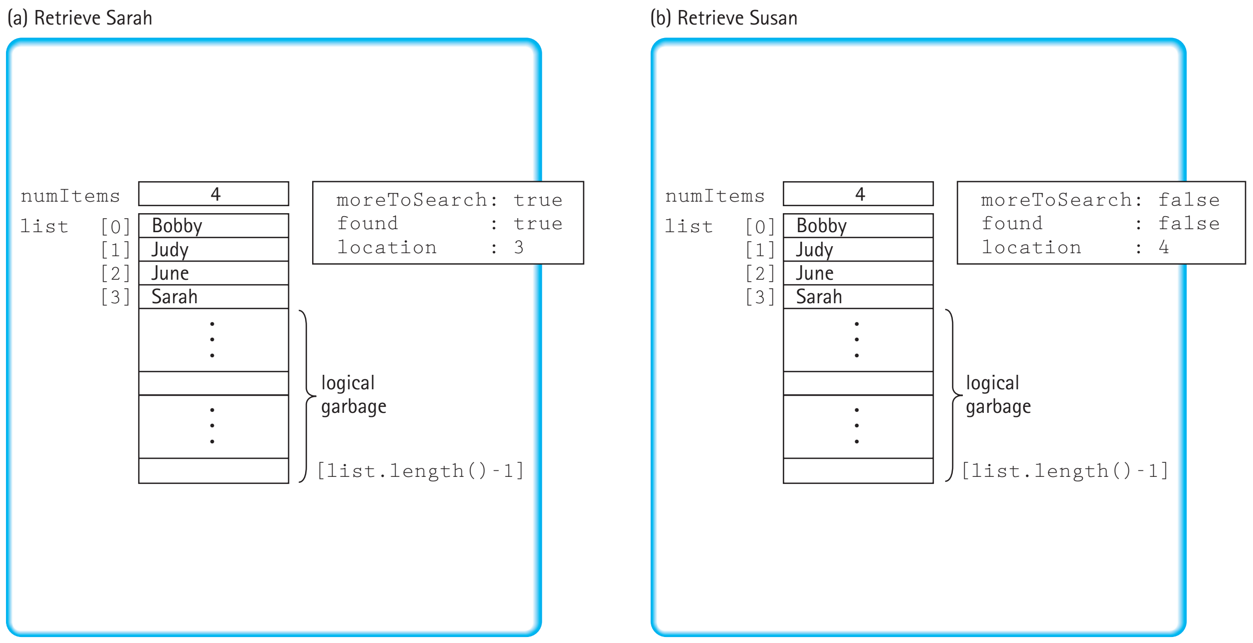


Figure 3.2 *Retrieving an item in an unsorted list.*

Before we code this algorithm, let's look at the cases where we find the item on the list and where we examine last.info() without finding it. We represent these cases in Figure 3.2 in an Honor Roll list. We first retrieve Sarah (see Figure 3.2(a)). Sarah is on the list, so when the search is completed, `moreToSearch` is `true`, `found` is `true`, and `location` is 3. The loop is exited because `found` became `true` when `item` was equal to the contents of location 3. Next, we retrieve Susan (see Figure 3.2(b)). Susan is not on the list, so when the search is completed `moreToSearch` is `false`, `found` is `false`, and `location` is equal to `numItems`. The loop is exited because `moreToSearch` became `false` after we examined the last information on the list.

Now we are ready to code the algorithm replacing the general design notation with the equivalent array notation. The substitutions are straightforward except for initializing `location` and determining whether we have examined last.info(). To initialize `location` in an array-based implementation in Java, we set it to 0. We know we have not examined last.info() as long as `location` is less than `numItems`. Be careful: Because Java indexes arrays from 0, the last item on the list is at index `numItems - 1`. Here is the coded algorithm.

```
public boolean isThere (String item)
// Returns true if item is on this list, otherwise returns false
{
  boolean moreToSearch;
  int location = 0;
  boolean found = false;
```

```
    moreToSearch = (location < numItems);

    while (moreToSearch && !found)
    {
      if (item.compareTo(list[location]) == 0)   // If they match
        found = true;
      else
      {
        location++;
        moreToSearch = (location < numItems);
      }
    }

   return found;
  }
```

insert Operation
Because the list elements are not sorted by value, we can put the new item anywhere. A straightforward strategy is to place the item in the `numItems` position and increment `numItems`.

insert (item)

```
Set numItems.info() to copy of item
Increment numItems
```

This algorithm is translated easily into Java.

```
  public void insert (String item)
  // Adds a copy of item to this list
  {
    list[numItems] = new String(item);
    numItems++;
  }
```

delete Operation
The `delete` method takes an item whose value indicates which item to delete. There are clearly two parts to this operation: finding the item to delete and removing it. We can use the `isThere` algorithm to search the list. When `compareTo` returns a nonzero value, we increment `location`; when `compareTo` returns 0, we exit the loop and remove the element. Because the preconditions for `delete` state that an item with the same key is definitely on the list, we do not need to test for reaching the end of the list.

How do we remove the element from the list? Let's look at the example in Figure 3.3. Removing Sarah from the list is easy, for hers is the last element on the list (see Figures 3.3a and 3.3b). If Bobby is deleted from the original list, however, we need to move up all the elements that follow to fill in the space—or do we? See Figure 3.3(c). If the list is sorted by value, we would have to move all the elements up as shown in Figure 3.3(c), but because the list is unsorted, we can just swap the item in the `numItems - 1` position with the item being deleted (see Figure 3.3(d)). In an array-based implementation, we do not actually remove the element; instead, we cover it up with the elements that previously followed it (if the list is sorted) or the element in the last position (if the list is unsorted). Finally, we decrement `numItems`.

```
public void delete (String item)
// Deletes the element that matches item from this list
{
  int location = 0;

  while (item.compareTo(list[location]) != 0)
    location++;

  list[location] = list[numItems - 1];
  numItems--;
}
```

Iterator Operations

The `reset` method is analogous to the Open operation for a file in which the file pointer is positioned at the beginning of the file so that the first input operation accesses the first component of the file. Each successive call to an input operation gets the next item in the file. Therefore, `reset` must initialize `currentPos` to indicate the first item on the list.

The `getNextItem` operation provides access to the next item on the list by returning currentPos.info( ) and incrementing `currentPos`. To do this, it must first "record" currentPos.info( ), then increment `currentPos`, and finally return the recorded information.

reset

```
Initialize currentPos to position of first list element
```

getNextItem: returns String

```
Set next to currentPos.info()
Set currentPos to currentPos.next()
return copy of next
```

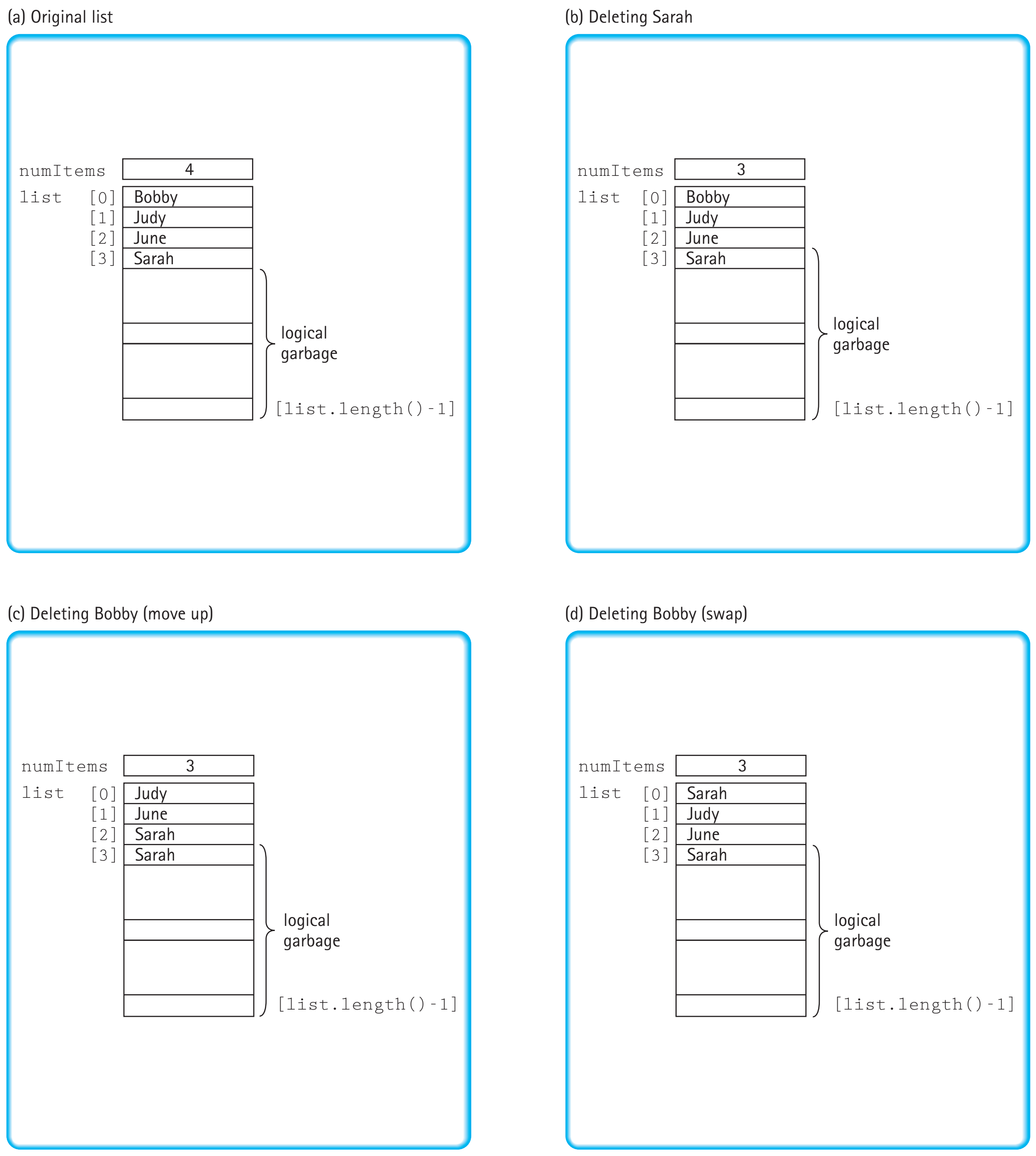


Figure 3.3 *Deleting an item in an unsorted list*

The `currentPos` value always indicates the next item to be processed in an iteration. To be safe, we decided to reset it automatically, in the `getNextItem` method, when the end of the list is reached. Therefore, there are two places where `currentPos` can be set to 0: in the `reset` method, and in the `getNextItem` method when the end of the list is reached. The code for the iteration operations is as follows:

```
public void reset()
// Initializes current position for an iteration through this list
{
  currentPos  = 0;
}

public String getNextItem ()
// Returns copy of the next element on this list
// And advances the current position
{
  String next = list[currentPos];
  if (currentPos == numItems-1)
    currentPos = 0;
  else
    currentPos++;
  return new String(next);
}
```

The `getNextItem` method could also be implemented using the modulus operation:

```
currentPos = (currentPos++) % (numItems - 1);
```

The `getNextItem` method returns a `String` variable. That means that it returns a reference to a string object. Notice that we have elected to create a new string object using the `String` class's copy constructor, and to return a reference to the new object, rather than a reference to the string object that is actually on the list. As we stated before, we implement our lists "by copy." Why did we do this? The answer is that we wish to maintain information hiding. If we return a reference into the list, we have given the application an alias of a hidden list element. So, rather than do that, we create a copy of the string, and return a reference to the copy. The list user is never allowed to directly see or manipulate the contents of the list. These details of the list implementation are encapsulated by the ADT.

In this case we are being overly protective; since strings are immutable objects there would be no potential harm in returning a reference to the actual string that is on the list. The application program cannot change the string, so in this case the work of copying the list object is unnecessary. Nevertheless, we wish to emphasize the need for care when returning values from within our ADTs. As mentioned previously, in Chapter 4 we follow the alternate approach, namely, returning references to the objects contained in our ADTs, and consider the strengths and drawbacks of each approach.

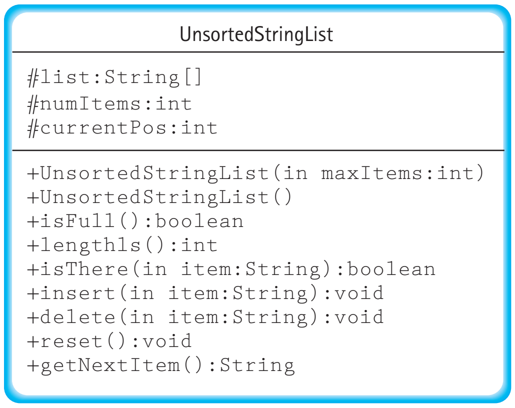


Figure 3.4 *UML diagram of* `UnsortedStringList`

Consider how the application programmer might use the list iteration methods. The programmer can use the length of the list to control a loop asking to see each item in turn. What happens if the program inserts or deletes an item in the middle of an iteration? Nothing good, you can be sure! Adding and deleting items changes the length of the list, making the termination condition of the iteration-counting loop invalid. Depending on whether an addition or deletion occurs before or after the iteration point, our iteration loop could end up skipping or repeating items.

We have several choices of how to handle this possibly dangerous situation. The list can throw an exception, the list can reset the current position when inserting or deleting, or the list can disallow transformer operations while an iteration is taking place. We choose the latter here by way of a precondition in the documentation.

The UML class diagram in Figure 3.4 represents our `UnsortedStringList` implementation.

Test Plan

To test our Unsorted List ADT, we create a test driver program similar to the one we created at the end of Chapter 1 to test the IncDate ADT. That test driver accepted a sequence of instructions from an input file that indicated which method of `IncDate` to invoke next. The test input also included any parameter values required by the `IncDate` methods. Results of the method invocations were printed to an output file. Meanwhile, a final count of the number of test cases was indicated in an output window.

As we planned when we created that test driver, it is not difficult to transform it into a test driver for a different ADT. To use it to test our Unsorted List ADT, we simply change the value assigned to the `testname` variable near the start of the program, change the declaration of the variables to appropriate ones for testing our list ADT, and rewrite the sequence of *if-else* statements to invoke and report on the list methods instead of the date methods.

We do not go into all the details of the code for the test driver. Note that since there are two constructors for the Unsorted List we must assign them two separate "code names" for our test input file. We simply chose to use "UnsortedStringList1" and "UnsortedStringList2". Here is the beginning of the main processing loop within the test driver:

```
// Process commands
while(!command.equals("quit"))
  {
    if (command.equals("UnsortedStringList1"))
  {
    size = Integer.parseInt(dataFile.readLine());
    list = new UnsortedStringList(size);
    outFile.println("The list is instantiated with size " + size);
  }
  else
  if (command.equals("UnsortedStringList2"))
  {
    list = new UnsortedStringList();;
    outFile.println("The list is instantiated with default size");
  }
  else
  if (command.equals("isFull"))
  {
    outFile.println("The list is full is " + list.isFull());
  }
   .
   .
   .
```

You can study the entire `TDUnsortedStringList.java` program (it's on our web site). What is important for us now is planning how to use the test driver to test our ADT.

The constructors `UnsortedStringList (int maxItems)` and `UnsortedString-List ()` can be exercised throughout our tests every time we create an `Unsorted StringList` object.

`lengthIs`, `insert`, and `delete` can be tested together. That is, we insert several items and check the length; we delete several items and check the length. How do we know that `insert` and `delete` work correctly? We can make calls to the `reset` and `getNextItem` methods to examine the structure of the list; a good approach would be to use `reset` and `getNextItem` to create a "print list" test method (such as defined in the application-level subsection), that could be called many times during the testing process. A `PrintList` method is included in the `TDUnsortedStringList.java` program.

To test the `isFull` operation, we can instantiate a list of size 5, insert four items and print the result of the test, and then insert the fifth item and print the result of the test. To test `isThere`, we must search for items that we know are on the list and for items that we know are not on the list.

How do we organize our test plan? We should classify our test possibilities. For example, an item can be in the first position on the list, in the last position on the list, or somewhere else on the list. So we must be sure that our `delete` can correctly delete items in these positions. We must also check that `isThere` can find items in these same positions. We should also check the `lengthIs` method at the boundary cases of an empty list and a full list. Notice that this test plan is mostly a black-box strategy. We are looking at the list as described in the interface, not in the code.

These observations are summarized in the following test plan, which concentrates on the observer methods and the `insert` method. To be complete the plan must be expanded to use both constructors, to test the `delete` method, to test various combinations of `insert` and `delete`, and, if program robustness is desired, to test how the software responds to situations precluded by the method preconditions—for example, insertion into a full list. The tests are shown in the order in which they should be implemented.

Operation to be Tested and Description of Action	Input Values	Expected Output
`UnsortedStringList`		
print `lengthIs`		0
print `isFull`		false
print `isThere("Tom")`		false
Print List		empty list
`UnsortedStringList`	5	
`insert`	Tom	
print `lengthIs`		1
print `isFull`		false
print `isThere`	Tom	true
Print List		Tom
`insert`	Julie	
`insert`	Nora	
`insert`	Maeve	
print `lengthIs`		4
print `isFull`		false
print `isThere`	Tom	true
print `isThere`	Julie	true
print `isThere`	Maeve	true
print `isThere`	Kevin	false
Print List		Tom, Julie, Nora, Maeve
`insert`	Kevin	
print `lengthIs`		5
print `isFull`		true
print `isThere`	Tom	true
print `isThere`	Julie	true
print `isThere`	Kevin	true
Print List		Tom, Julie, Nora, Maeve, Kevin
etc.		

The file `testlist1.dat`, that accompanies the program file on our web site, provides an example of a set of test data. It is not a complete test. The file `testout1.dat` shows the results of the following program invocation:

```
java TDUnsortedStringList testlist1.dat testout1.dat
```

The key to properly testing any software is in the plan: It must be carefully thought out and it must be written. We have discussed the basic approach needed for testing the Unsorted List ADT, listed a partial test plan, and provided a test driver (in the file `TDUnsortedStringList.java`). We leave the creation of the complete written test plan as an exercise.

3.3 Abstract Classes

We have just completed the design and implementation of an Unsorted List ADT. In the next section we follow the same basic approach to create a Sorted List ADT. However, before we do that we take a look at Java's abstract class mechanism. We can use an abstract class to take advantage of the similarities between the Unsorted and Sorted List ADTS.

Relationship between Unsorted and Sorted Lists

Suppose you are given the task of creating a Sorted List ADT. The first step you might take is to identify the logical operations that you need to include. As you start to identify the operations, you might have the feeling that you have done this exercise before. Let's see, you'll need constructors to create your list. You'll need some way to put things onto the list, so you need an `insert` method. Of course, you might want to remove things from the list, so you need a `delete` method.

Sound familiar? As you think about it, you realize that all of the logical operations we defined for the Unsorted List ADT are also needed for your Sorted List ADT. The logical definition of those operations did not rely on whether or not the list was sorted. If you look at the Unsorted List ADT specification, you can see that the entire specification may be reused. The only changes that need to be made are to the preconditions and postconditions of the transformer methods `insert` and `delete`: They must specify that the list is sorted. `insert` and `delete` are the only two methods that affect the underlying ordering of the list items. The condition

"The list is sorted."

can be added to both their preconditions and postconditions, and you are all set.

Since you were able to reuse most of the specification of the Unsorted List ADT to specify your Sorted List ADT, maybe you can also reuse some of the implementation. In fact, assuming you again wish to use an array-based implementation, you can reuse the entire class except the implementations of the `insert` and `delete` methods. We look at the implementations of these methods in the next subsection. We also look at a variant of the isThere method. Although you can just reuse the isThere method of the Unsorted List ADT, we are able to create a more efficient version of the method under the assumption that the list is sorted.

Reuse Options

There are several ways we could reuse the code of the Unsorted List ADT to create the code for the Sorted List ADT. Let's look at three approaches.

Cut and Paste

We could create a Sorted List class completely independent of the Unsorted List ADT class. Just create a new file called SortedStringList.java, "cut and paste" the code that we are able to reuse into the new file, rename the constructors to match the file name, and create the new code for the three methods that we want to change. Once we are finished there is no longer any formal link between the two classes: Cut and paste, direct inheritance, and abstract classes.

However, this lack of connection between the two classes can be detrimental. Consider, for example, if someone using the Sorted List class discovers an error in the getNexItem method. Suppose they fix the method but do not realize that a "copy" of the method exists in another class? This means that although a bug has been detected, and a solution devised and implemented, the same bug is still plaguing another class. If we could somehow formally link the two classes together, so that the code for the common methods only appears in one place, then both classes would share any updates made to these methods.

Direct Inheritance

Since the Sorted List ADT class can use several of the methods of the Unsorted List class, maybe we should make the former a subclass of the latter:

```
public class SortedStringList extends UnsortedStringList
. . .
```

Within the Sorted List class we can redefine the three methods that need to be changed. With this approach we do create a formal link between the two classes, and changes to the shared methods would affect both classes.

While this approach is probably better than the previous approach, it still has some problems. The main problem is that the inheritance relationship just doesn't make sense. Recall that in Chapter 1 we stated that the inheritance relationship usually represents an "is a" relationship. In the example in Chapter 1, an IncDate *is a* Date; an IncDate object was a special kind of Date object. Here, that relationship doesn't make sense. Saying that a sorted list *is a* unsorted list sounds like nonsense.

Just because the *is a* relationship does not make sense doesn't mean that you can't use inheritance. It does, however, often mean that using inheritance might lead to problems later. For example, due to Java's rules for assignment of object variables, it would be possible for an application program to include the following code, assuming that the Sorted List ADT inherited from the Unsorted List ADT:

```
UnsortedStringList unsorted;
SortedStringList sorted = new SortedStringList(10);
unsorted = sorted;
```

This creates the rather confusing situation in which an Unsorted List variable is referencing a Sorted List object. This is completely legal in the world of Java—and it usually makes sense if the *is a* relationship makes sense. But as you can see, in this case it seems illogical. So, although using inheritance solves the problems identified in the previous subsection, another approach might be more appropriate.

Abstract Classes

Java offers another construct, called an *abstract class*, which resolves the deficiencies of both of the previous approaches.

An *abstract method* is one that is declared without a method body. For the sake of this discussion, let's call a normal method that is declared with a body a *concrete method*.

We discussed abstract methods in Chapter 2 when we looked at the Java `interface` construct. You may recall that a Java interface was not allowed to contain any concrete methods; it could only contain abstract methods. An abstract class, on the other hand, can contain both concrete methods and abstract methods. It must contain at least one abstract method. To indicate that a class is abstract, we use the Java keyword `abstract` in its definition. You'll see an example of this syntax in the next subsection. An abstract class cannot be instantiated. It must be extended by another class, which provides the missing implementations of the abstract methods.

We previously pointed out that it does not make sense to say that a sorted list *is a* unsorted list. Similarly, it doesn't make sense to reverse that; it does not make sense to say an unsorted list *is a* sorted list. What then is the relationship between a sorted list and an unsorted list? Easy, they are both lists! We can use an abstract class to model this relationship.

We first create an abstract list class; its concrete methods provide the operations that our two list ADTs share in common and its abstract methods provide the operations that are not shared. We can then create two concrete classes that extend the abstract list class, one that implements an unsorted list and the other that implements a sorted list. With this approach we maintain the common code for the shared methods and we create a reasonable *is a* inheritance structure: an unsorted list is a list and a sorted list is a list.

An Abstract List Class

Our abstract list class is very straightforward. It is based on the `UnsortedStringList` class developed in the previous section. We simply change the name of the class, and the constructor, to StringList, add the keyword abstract to the header line, and remove the method bodies from the insert, delete, and isThere methods. We now declare these three methods as abstract, and end their declaration lines with a semicolon. You should also notice that we have only retained one of the constructors, the one which accepts an integer parameter maxItems. The other constructor is redundant in this scheme, as you see when we extend this class with the concrete classes in the next section. Finally, notice that we place StringList in the same package as we placed UnsortedStringList.

```
//----------------------------------------------------------------------------
// StringList.java             by Dale/Joyce/Weems                   Chapter 3
//
// Defines all constructs for an array-based list that do not depend
// on whether or not the list is sorted
//----------------------------------------------------------------------------

package ch03.stringLists;

public abstract class StringList
{
  protected String[] list;           // Array to hold this list's elements
  protected int numItems;            // Number of elements on this list
  protected int currentPos;          // Current position for iteration

  public StringList(int maxItems)
  // Instantiates and returns a reference to an empty list object
  // with room for maxItems elements
  {
    numItems = 0;
    list = new String[maxItems];
  }

  public boolean isFull()
  // Returns whether this list is full
  {
    return (list.length == numItems);
  }

  public int lengthIs()
  // Returns the number of elements on this list
  {
    return numItems;
  }
  public abstract boolean isThere (String item);
  // Returns true if item is on this list; otherwise, returns false

  public abstract void insert (String item);
  // Adds a copy of item to this list

  public abstract void delete (String item);
  // Deletes the element that matches item from this list.

  public void reset()
  // Initializes current position for an iteration through this list
  {
    currentPos  = 0;
  }

  public String getNextItem ()
  // Returns copy of the next element on this list
  {
    String next = list[currentPos];
    if (currentPos == numItems-1)
      currentPos = 0;
    else
      currentPos++;
    return new String(next);
  }
}
```

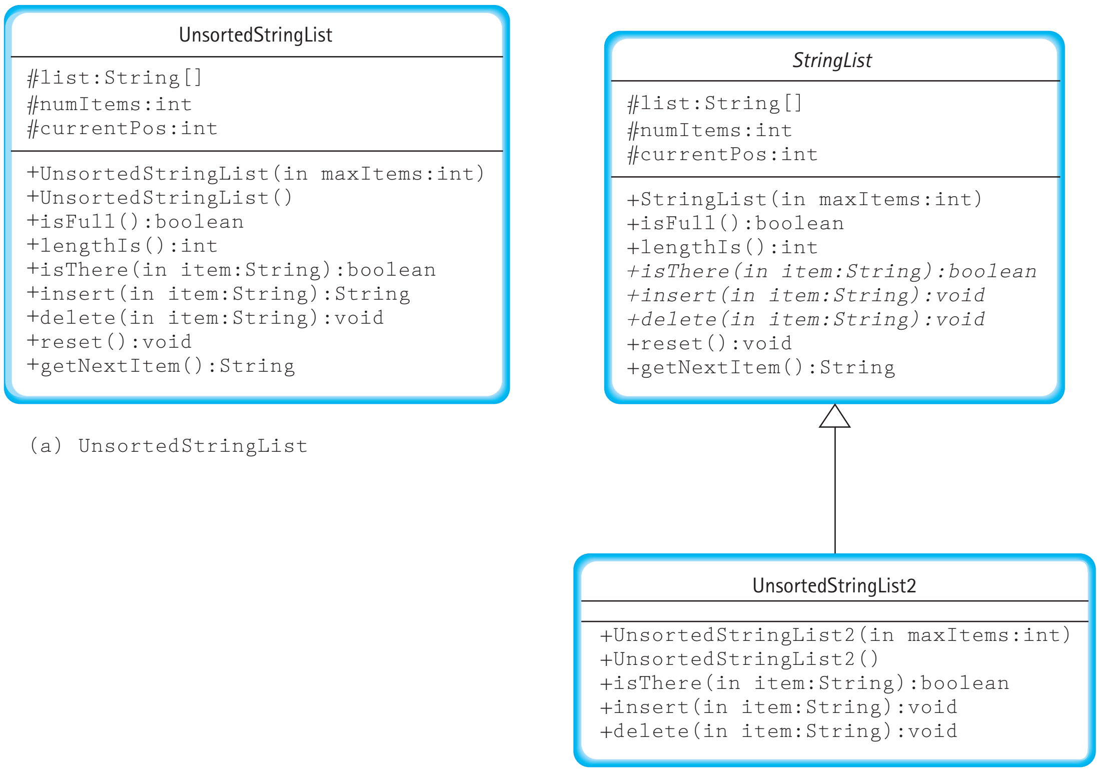
Extending the Abstract Class

Now we can create an Unsorted List ADT class by extending the abstract list class. To differentiate this Unsorted List class from the one developed in the previous section, we call it `UnsortedStringList2`. Since constructors cannot be inherited, we must implement our own constructors for this class. Notice how our code for the two constructors both use the single constructor provided in the abstract list class. Additionally, we must complete the definitions of the three abstract classes. We simply reuse the code from the previous implementations. The code for the new unsorted string list is shown below.

```
//----------------------------------------------------------------------------
// UnsortedStringList2.java        by Dale/Joyce/Weems              Chapter 3
//
// Completes the definition of the StringList class under the assumption
// that the list is not kept sorted
//----------------------------------------------------------------------------

package ch03.stringLists;
```

```
public class UnsortedStringList2 extends StringList
{
  public UnsortedStringList2(int maxItems)
  // Instantiates and returns a reference to an empty list object
  // with room for maxItems elements
  {
    super(maxItems);
  }

  public UnsortedStringList2()
  // Instantiates and returns a reference to an empty list object
  // with room for 100 elements
  {
    super(100);
  }

  public boolean isThere (String item)
  // Returns true if item is on this list; otherwise, returns false
  {
    boolean moreToSearch;
    int location = 0;
    boolean found = false;

    moreToSearch = (location < numItems);

    while (moreToSearch && !found)
    {
      if (item.compareTo(list[location]) == 0)  // if they match
        found = true;
      else
      {
        location++;
        moreToSearch = (location < numItems);
      }
    }

   return found;
  }

  public void insert (String item)
  // Adds a copy of item to this list
  {
    list[numItems] = new String(item);
    numItems++;
  }
  public void delete (String item)
  // Deletes the element that matches item from this list
  {
    int location = 0;

    while (item.compareTo(list[location]) != 0)
      location++;

    list[location] = list[numItems - 1];
    numItems--;
  }
}
```

The UML class diagram in Part (b) of Figure 3.5 models both the abstract `StringList` class and the `UnsortedStringList2` class. Note that the diagram displays the `isThere`, `insert`, and `delete` methods defined in the `StringList` class, and the name of the class itself, in an italic font to indicate that they are abstract classes. Part (a) of the diagram models our original `UnsortedStringList` class, to allow comparison.


UnsortedStringList
#list:String[]
#numItems:int
#currentPos:int
+UnsortedStringList(in maxItems:int)
+UnsortedStringList()
+isFull():boolean
+lengthIs():int
+isThere(in item:String):boolean
+insert(in item:String):String
+delete(in item:String):void
+reset():void
+getNextItem():String
StringList
#list:String[]
#numItems:int
#currentPos:int
+StringList(in maxItems:int)
+isFull():boolean
+lengthIs():int
+isThere(in item:String):boolean
+insert(in item:String):void
+delete(in item:String):void
+reset():void
+getNextItem():String
UnsortedStringList2
+UnsortedStringList2(in maxItems:int)
+UnsortedStringList2()
+isThere(in item:String):boolean
+insert(in item:String):void
+delete(in item:String):void

(a) `UnsortedStringList`

(b) `UnsortedStringList2`

Figure 3.5 *UML diagrams for our list implementations*

3.4 Abstract Data Type Sorted List

At the beginning of this chapter, we said that a list is a linear sequence of items; from any item (except the last) you can access the next one. We looked at the specifications and implementation for the operations that manipulate a list and guarantee this property.

We now want to add an additional property: The key member of any item (except the last) comes before the key member of the next one. We call a list with this property a *sorted list.*

Logical Level

When we defined the specifications for the Unsorted List ADT, we made no requirements with respect to the order in which the list elements are stored and maintained. Now, we have to change the specifications to guarantee that the list is sorted. As was noted in the section Relationship between Unsorted and Sorted Lists of Section 3.3, we must add preconditions and postconditions to those operations for which order is relevant. The only ones that must be changed are `insert` and `delete`.

We call our new class the `SortedStringList` class. We must define new constructors, since their names are directly related to the name of the class.

Sorted List ADT Specification (partial)

Structure:

The list elements are Strings. The list contains unique elements, i.e., no duplicate elements as defined by the key of the list. The strings are kept in alphabetical order. The list has a special property called the current position—the position of the next element to be accessed by `getNextItem` during an iteration through the list. Only `reset` and `getNextItem` affect the current position.

Definitions (provided by user):

maxItems:	An integer specifying the maximum number of items to be on this list.

Operations (provided by Sorted List ADT):

void SortedStringList (int maxItems)

Effect:	Instantiates this list with capacity of `maxItems` and initializes this list to empty state.
Precondition:	`maxItems` > 0
Postcondition:	This list is empty.

void SortedStringList ( )

Effect:	Instantiates this list with capacity of 100 and initializes this list to empty state.
Postcondition:	This list is empty.

void insert (String item)

Effect:	Adds `item` to list.
Preconditions:	List is not full.
	`item` is not on the list.
	List is sorted.
Postconditions:	`item` is on the list.
	List is still sorted

void delete (String item)

Effect:	Deletes the element whose key matches `item`'s key.
Preconditions:	One and only one element in list has a key matching `item`'s key.
	List is sorted.
Postconditions:	No element in list has a key matching the argument `item`'s key.
	List is still sorted.
	The remaining operations use the same definitions as the Unsorted List ADT.

Application Level

The application level for the Sorted List ADT is the same as for the Unsorted List ADT. As far as the user is concerned, the interfaces are the same. The only functional difference is that when `getNextItem()` is called in the Sorted List ADT, the element returned is the next one in order by key.

Implementation Level

We continue to use the generic list design terminology, created to describe the algorithms for the Unsorted List ADT operations, to describe the algorithms in this section.

insert Operation

To add an element to a sorted list, we must first find the place where the new element belongs, which depends on the value of its key. We use an example to illustrate the insertion operation. Let's say that Becca has made the Honor Roll. To add the element Becca to the sorted list pictured in Figure 3.6(a), maintaining the alphabetic ordering, we must accomplish three tasks:

1. Find the place where the new element belongs.
2. Create space for the new element.
3. Put the new element on the list.

The first task involves traversing the list comparing the new item to each item on the list until we find an item where the new item (in this case, Becca) is less. Recall from Chapter 2 that the `String` method `compareTo` takes a string as a parameter and returns 0 if the parameter string and the object string are equal, returns a positive integer if the parameter string is "less than" the object string, and returns a negative integer if the parameter string is "greater than" the object string. Therefore, we set `moreToSearch` to `false` when we reach a point where `item.compareTo(location.info())` is negative. At this point, `location` is where the new item should go (see Figure 3.6b). If we don't find a place

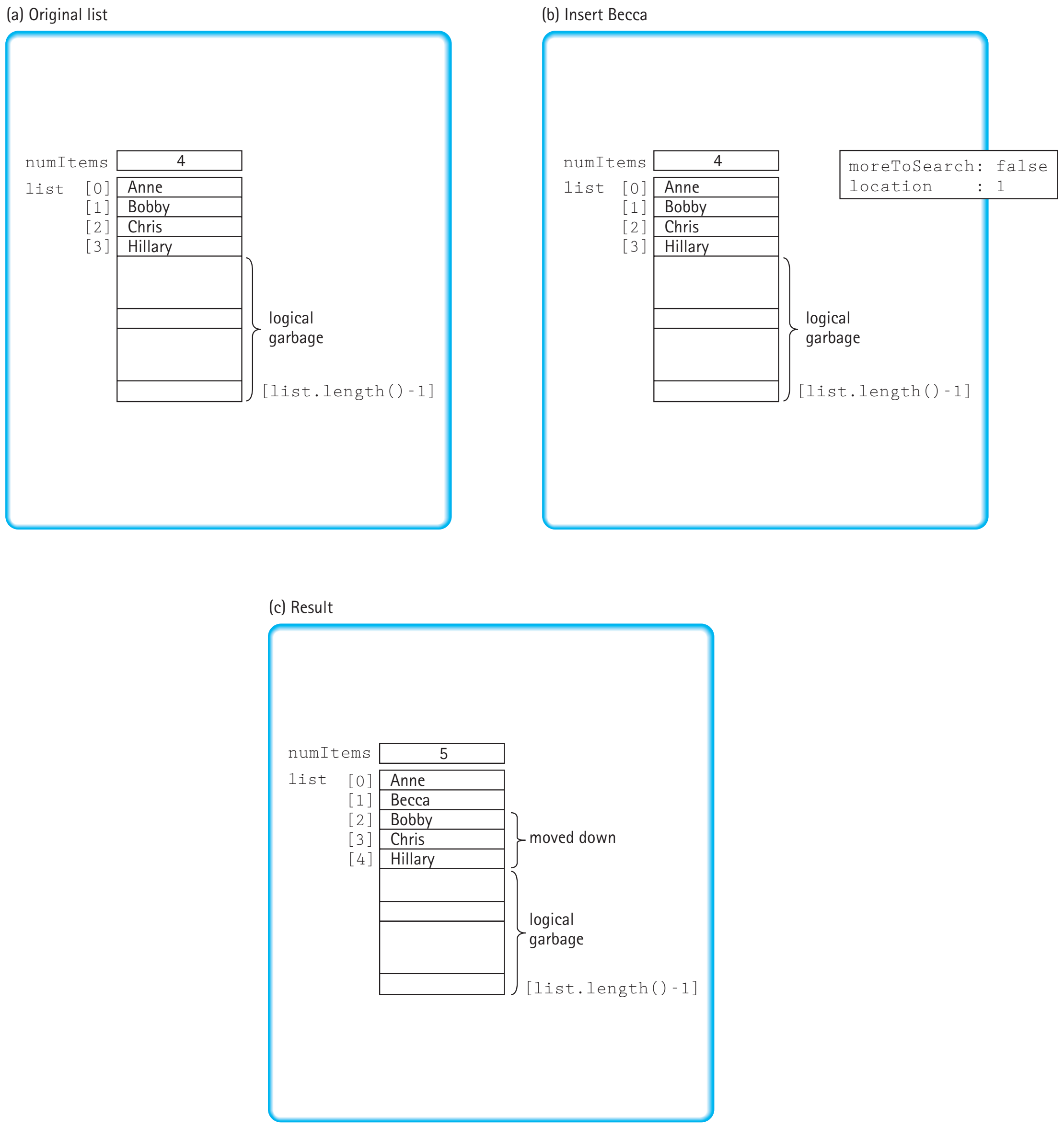


Figure 3.6 *Inserting into a sorted list*

where `item.compareTo(location.info())` is negative, then the item should be put at the end of the list. This is true when `location` equals `numItems`.

Now that we know where the element belongs, we need to create space for it. Because the list is sequential, Becca must be put into the list at location.info( ). But this position may be occupied. To "create space for the new element," we must move down all the list elements that follow it, from `location` through `numItems - 1`. Now we just assign `item` to location.info( ) and increment `numItems`. Figure 3.6(c) shows the resulting list.

Let's summarize these observations in algorithmic form before we write the code.

insert (item)

```
Initialize location to position of first element
Set moreToSearch to (have not examined last.info())
while moreToSearch
  if (item.compareTo(location.info()) < 0)
    Set moreToSearch to false
  else
    Set location to location.next()
    Set moreToSearch to (have not examined last.info())
for index going from numItems DOWNTO location + 1
  Set index.info() to (index-1).info()
Set location.info() to copy of item
Increment numItems
```

Remember that the preconditions on `insert` state that item does not exist on the list, so we do not need to check whether the `compareTo` method returns a zero. Translating the design notation into the array-based implementation gives us the following method.

```
public void insert (String item)
// Adds a copy of item to this list
{
  int location = 0;
  boolean moreToSearch = (location < numItems);

  while (moreToSearch)
  {
    if (item.compareTo(list[location]) < 0)   // Item is less
      moreToSearch = false;
    else                                       // Item is more
    {
      location++;
      moreToSearch = (location < numItems);
    }
  }

  for (int index = numItems; index > location; index--)
    list[index] = list[index - 1];

  list[location] = new String(item);
  numItems++;
}
```

Does this method work if the new element belongs at the beginning or end of the list? Draw a picture to see how the method works in each of these cases.

delete Operation

When discussing the method `delete` for the Unsorted List ADT, we commented that if the list is sorted, we would have to move the elements up one position to cover the one being removed. Moving the elements up one position is the mirror image of moving the elements down one position. The loop control for finding the item to delete is the same as for the unsorted version.

delete (item)

```
Initialize location to position of first element
while (item.compareTo(location.info()) != 0)
    Set location to location.next()
for index going from location + 1 TO numItems - 1
    Set (index-1).info() to index.info()
Decrement numItems
```

Examine this algorithm carefully and convince yourself that it is correct. Try cases where you are deleting the first item and the last one.

```
public void delete (String item)
// Deletes the element that matches item from this list
{
  int location = 0;
  while (item.compareTo(list[location]) != 0)    // while not a match
    location++;

  for (int index = location + 1; index < numItems; index++)
    list[index - 1] = list[index];

  numItems--;
}
```

Improving the isThere Operation

If the list is not sorted, the only way to search for an item is to start at the beginning and look at each element on the list, comparing the key member of the item for which we are searching to the key member of each element on the list in turn. This was the algorithm used in the `isThere` operation in the Unsorted List ADT.

If the list is sorted by key value, there are two ways to improve the searching algorithm. The first way is to stop searching when we pass the place where the item would be if it were there. Look at Figure 3.7(a). If you are searching for Chris, a comparison with Judy would show that Chris is less, that is, the `compareTo` method returns a positive integer. This means that you have passed the place where Chris would be if it were there. At this point you can stop and return `found` as `false`. Figure 3.7(b) shows what happens when you are searching for Susy: `location` is equal to 4, `moreToSearch` is `false`, and `found` is `false`. In this case the search ends because there is nowhere left to look.

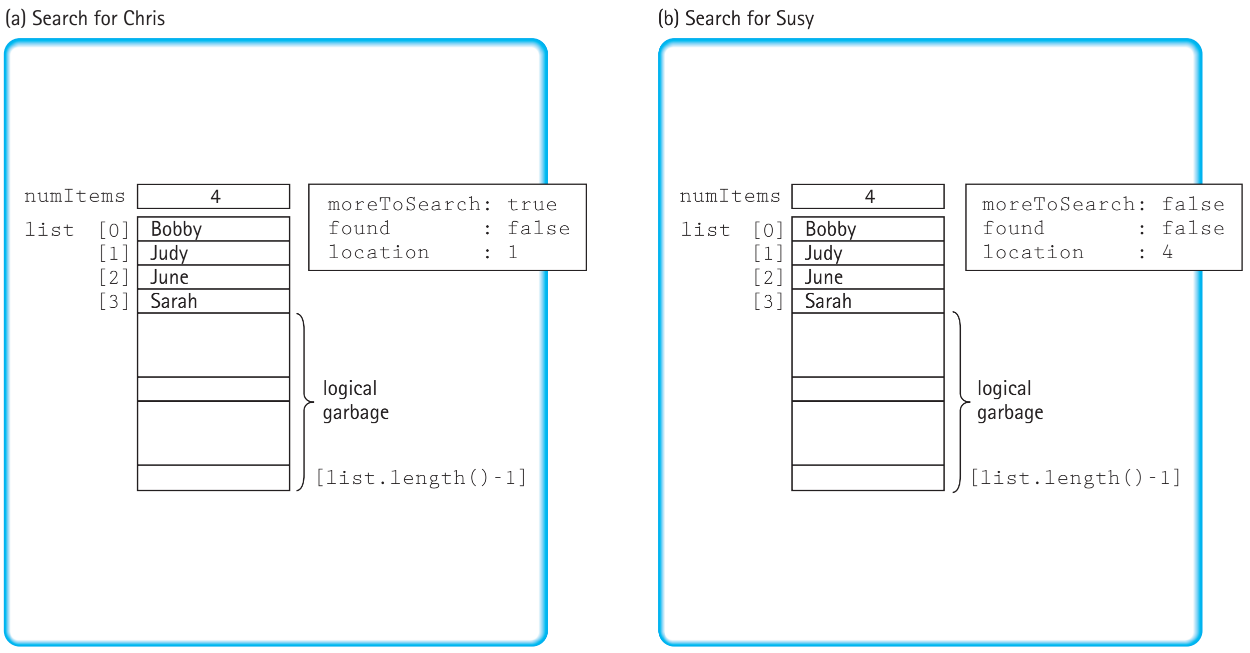


Figure 3.7 *Retrieving in a sorted list*

If the item we are looking for is on the list, the search is the same for the unsorted list and the sorted list. It is when the item is not there that this algorithm is better. We do not have to search all of the elements to determine that the one we want is not there. The second way to improve the algorithm, using a binary search approach, helps in both the case when the item is on the list and the case when the item is not on the list.

Binary Search Algorithm

Think of how you might go about finding a name in a phone book, and you can get an idea of a faster way to search. Let's look for the name "David." We open the phone book to the middle and see that the names there begin with M. M is larger than (comes after) D, so we search the first half of the phone book, the section that contains A to M. We turn to the middle of the first half and see that the names there begin with G. G is larger than D, so we search the first half of this section, from A to G. We turn to the middle page of this section, and find that the names there begin with C. C is smaller than D, so we search the second half of this section—that is, from C to G—and so on, until we are down to the single page that contains the name "David." This algorithm is illustrated in Figure 3.8.

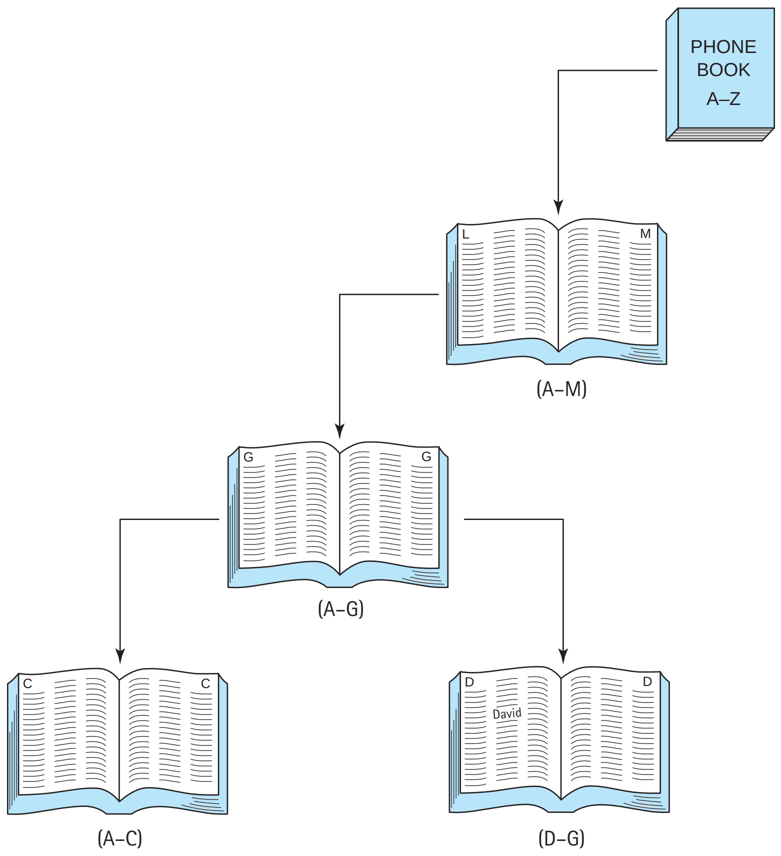


Figure 3.8 *A binary search of the phone book*

The algorithm presented here depends directly on the array-based implementation of the list. This algorithm cannot be implemented with the linked implementation presented in Chapter 5. Therefore, in discussing this algorithm we abandon our generic list design terminology in favor of using array-related terminology.

We begin our search with the whole list to examine; that is, our current search area goes from `list[0]` through `list[numItems - 1]`. In each iteration, we split the current search area in half at the midpoint, and if the item is not found there, we search the appropriate half. The part of the list being searched at any time is the current search area. For instance, in the first iteration of the loop, if a comparison shows that the item comes before the element at the midpoint, the new current search area goes from index 0 through `midpoint - 1`. If the item comes after the element at the midpoint, the new current search area goes from index `midpoint + 1` through `numItems - 1`. Either way, the current search area has been split in half. It looks as if we can keep track of the boundaries of the current search area with a pair of indexes, `first` and `last`. In each iteration of the loop, if an element with the same key as `item` is not found, one of these indexes is reset to shrink the size of the current search area.

How do we know when to quit searching? There are two possible terminating conditions: `item` is not on the list and item has been found. The first terminating condition occurs when there's no more to search in the current search area. Therefore, we only continue searching if `(first <= last)`. The second terminating condition occurs when `item` has been found.

isThere (item): returns boolean

```
Set first to 0
Set last to numItems – 1
Set found to false
Set moreToSearch to (first <= last)
while moreToSearch AND NOT found
  Set midPoint to (first + last) / 2
  compareResult = item.compareTo(midPoint.info())
  if compareResult == 0
    Set found = true
  else if compareResult < 0
    Set last to midPoint – 1
    Set moreToSearch to (first <= last)
  else
    Set first to midPoint + 1
    Set moreToSearch to (first <= last)
return found
```

Notice that when we look in the lower half or upper half, we can ignore the midpoint because we know it is not there. Therefore, `last` is set to `midPoint - 1`, or `first` is set to `midPoint + 1`. The coded version of our algorithm follows.

```
public boolean isThere (String item)
// Returns true if item is on this list; otherwise, returns false
{
  int compareResult;
  int midPoint;
  int first = 0;
  int last = numItems - 1;
  boolean moreToSearch = (first <= last);
  boolean found = false;

  while (moreToSearch && !found)
  {
    midPoint = (first + last) / 2;
    compareResult = item.compareTo(list[midPoint]);

    if (compareResult == 0)
      found = true;
    else if (compareResult < 0)  // Item is less than element at location
    {
      last = midPoint - 1;
      moreToSearch = (first <= last);
    }
    else                         // Item is greater than element at location
    {
      first = midPoint + 1;
      moreToSearch = (first <= last);
    }
  }

 return found;
}
```

Let's do a walk-through of the binary search algorithm. The item being searched for is "bat". Figure 3.9 (a) shows the values of `first`, `last`, and `midpoint` during the first iteration. In this iteration, "bat" is compared with "dog," the value in `list[midpoint]`. Because "bat" is less than (comes before) "dog," `last` becomes `midpoint - 1` and `first` stays the same. Figure 3.9(b) shows the situation during the second iteration. This time, "bat" is compared with "chicken," the value in `list[midpoint]`. Because "bat" is less than (comes before) "chicken," `last` becomes `midpoint - 1` and `first` again stays the same.

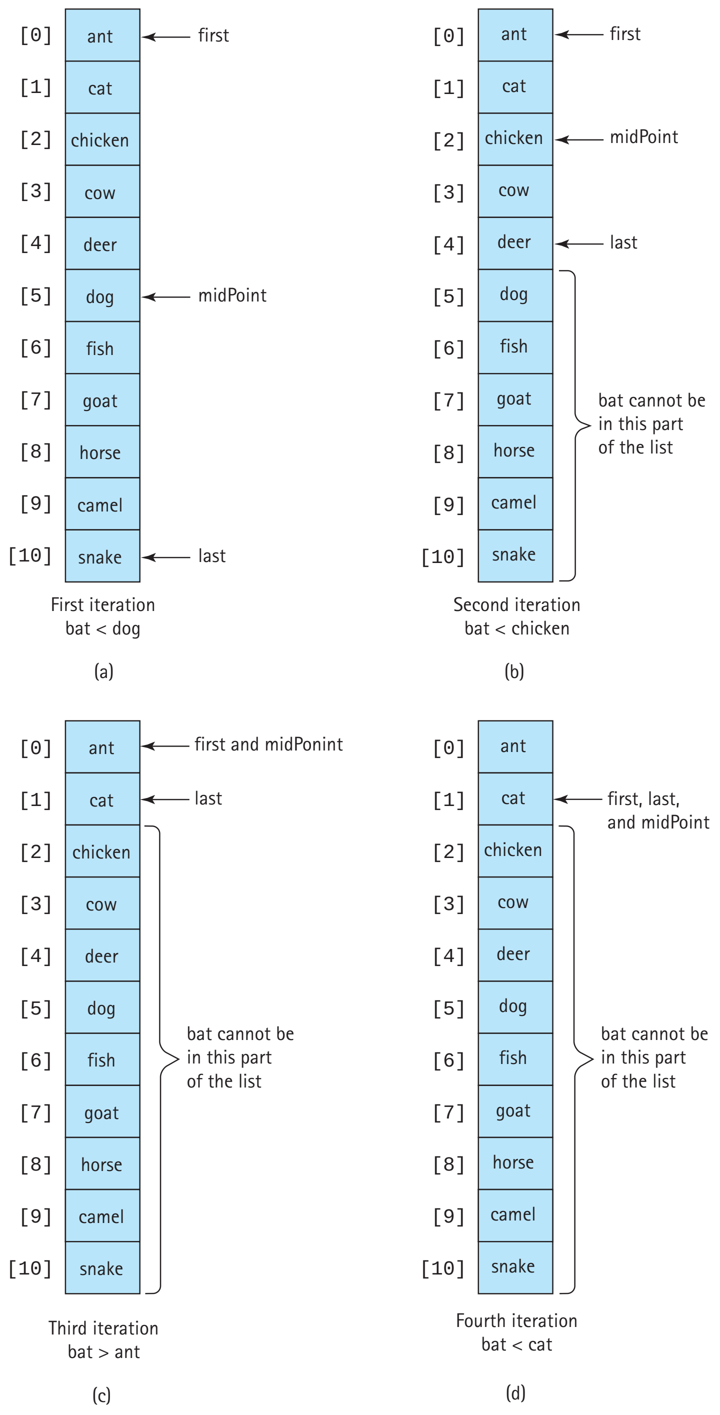


Figure 3.9 *Trace of the binary search algorithm*

In the third iteration (Figure 3.9c), `midpoint` and `first` are both 0. The item "bat" is compared with "ant," the item in `list[midpoint]`. Because "bat" is greater than (comes after) "ant," `first` becomes `midpoint + 1`. In the fourth iteration (Figure 3.9d), `first`, `last`, and `midpoint` are all the same. Again, "bat" is compared with the item in `list[midpoint]`. Because "bat" is less than "cat," `last` becomes `midpoint -1`. Now that `last` is less than `first`, the process stops; `found` is `false`.

The binary search is the most complex algorithm that we have examined so far. The following table shows `first`, `last`, `midpoint`, and `list[midpoint]` for searches of the items "fish," "snake," and "zebra," using the same data as in the previous example. Examine the results of Table 3.1 carefully.

Notice that the loop never executes more than four times. It never executes more than four times in a list of 11 components because the list is being cut in half each time through the loop. Table 3.2 compares a linear search and a binary search in terms of the average number of iterations needed to find an item.

If the binary search is so much faster, why not use it all the time? It is certainly faster in terms of the number of times through the loop, but more computations are executed within the binary search loop than in the other search algorithms. So if the number of components on the list is small (say, under 20), linear search algorithms are faster because they perform less work at each iteration. As the number of components on the list increases, the binary search algorithm becomes relatively more efficient. Remember, however, that the binary search requires the list to be sorted and sorting takes time.

The UML diagram for the `SortedStringList` class is displayed in Figure 3.10, along with the diagrams for the previous list implementations for comparison purposes.

Table 3.1 *Trace of binary search algorithm*

Iteration	first	last	midPoint	list[midPoint]	Terminating Condition
item: fish					
First	0	10	5	dog	
Second	6	10	8	horse	
Third	6	7	6	fish	found is true
item: snake					
First	0	10	5	dog	
Second	6	10	8	horse	
Third	9	10	9	camel	
Fourth	10	10	10	snake	found is true
item: zebra					
First	0	10	5	dog	
Second	6	10	8	horse	
Third	9	10	9	camel	
Fourth	10	10	10	snake	
Fifth	11	10			last < first

Table 3.2 *Comparison of linear and binary search*

	Average Number of Iterations	
Length	Linear Search	Binary Search
10	5.5	2.9
100	50.5	5.8
1,000	500.5	9.0
10,000	5000.5	12.0

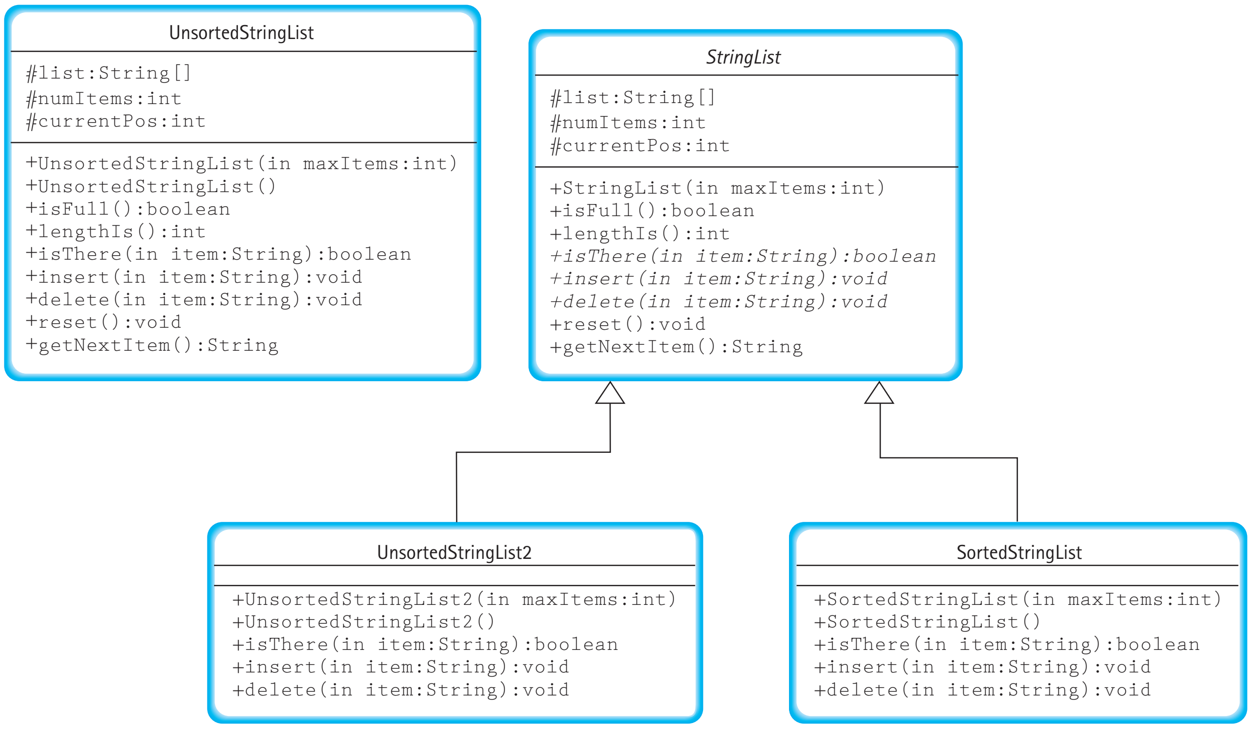

UnsortedStringList
#list:String[]
#numItems:int
#currentPos:int
+UnsortedStringList(in maxItems:int)
+UnsortedStringList()
+isFull():boolean
+lengthIs():int
+isThere(in item:String):boolean
+insert(in item:String):void
+delete(in item:String):void
+reset():void
+getNextItem():String
StringList
#list:String[]
#numItems:int
#currentPos:int
+StringList(in maxItems:int)
+isFull():boolean
+lengthIs():int
+isThere(in item:String):boolean
+insert(in item:String):void
+delete(in item:String):void
+reset():void
+getNextItem():String
UnsortedStringList2
+UnsortedStringList2(in maxItems:int)
+UnsortedStringList2()
+isThere(in item:String):boolean
+insert(in item:String):void
+delete(in item:String):void
SortedStringList
+SortedStringList(in maxItems:int)
+SortedStringList()
+isThere(in item:String):boolean
+insert(in item:String):void
+delete(in item:String):void

Figure 3.10 *UML diagrams for our list implementations*

Test Plan

We can use the same test plan that we used for the unsorted list, with the expected outputs changed to reflect the ordering. However, we should add some test cases to explicitly address the fact that the list is sorted. For example, we should insert a sequence of strings in reverse alphabetical order and check if the ADT correctly orders them. Note that the sorted list implementation described in this section can be found in the file `SortedStringList.java` on our web site.

3.5 Comparison of Algorithms

As we have shown in this chapter, there is more than one way to solve most problems. If you were asked for directions to Joe's Diner (see Figure 3.11), you could give either of two equally correct answers:

1. "Go east on the big highway to the Y'all Come Inn, and turn left."
2. "Take the winding country road to Honeysuckle Lodge, and turn right."

The two answers are not the same, but because following either route gets the traveler to Joe's Diner, both answers are functionally correct.

If the request for directions contained special requirements, one solution might be preferable to the other. For instance, "I'm late for dinner. What's the quickest route to

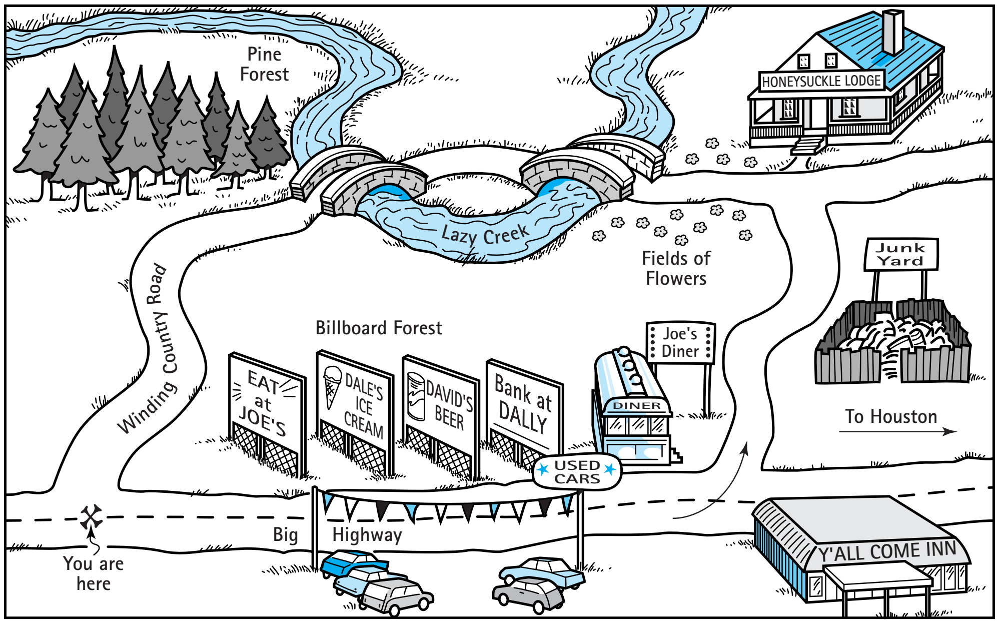


Figure 3.11 *Map to Joe's Diner*

Joe's Diner?" calls for the first answer, whereas "Is there a scenic road that I can take to get to Joe's Diner?" suggests the second. If no special requirements are known, the choice is a matter of personal preference—which road do you like better?

In this chapter, we have presented many algorithms. How we choose between two algorithms that do the same task often depends on the requirements of a particular application. If no relevant requirements exist, the choice may be based on the programmer's own style.

Often the choice between algorithms comes down to a question of efficiency. Which one takes the least amount of computing time? Which one does the job with the least amount of work? We are talking here of the amount of work that the computer does. Later we also compare algorithms in regard to how much work the programmer does. (One is often minimized at the expense of the other.)

To compare the work done by competing algorithms, we must first define a set of objective measures that can be applied to each algorithm. The analysis of algorithms is an important area of theoretical computer science; in advanced courses students undoubtedly see extensive work in this area. In this text you learn about a small part of this topic, enough to let you determine which of two algorithms requires less work to accomplish a particular task.

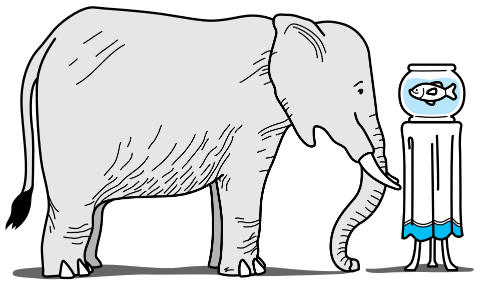
How do programmers measure the work that two algorithms perform? The first solution that comes to mind is simply to code the algorithms and then compare the execution times for running the two programs. The one with the shorter execution time is clearly the better algorithm. Or is it? Using this technique, we really can determine only that program A is more efficient than program B on a particular computer at a particular time. Execution times are specific to a *particular computer*, since different computers run at different speeds. Sometimes they are dependent on what else the computer is doing in the background, for example if the Java run-time engine is performing garbage collection, it can affect the execution time of the program. Of course, we could test the algorithms on many possible computers at various times, but that would be unrealistic and too specific (new computers are becoming available all the time). We want a more general measure.

A second possibility is to count the number of instructions or statements executed. This measure, however, varies with the programming language used, as well as with the style of the individual programmer. To standardize this measure somewhat, we could count the number of passes through a critical loop in the algorithm. If each iteration involves a constant amount of work, this measure gives us a meaningful yardstick of efficiency.

Another idea is to isolate a particular operation fundamental to the algorithm and count the number of times that this operation is performed. Suppose, for example, that we are summing the elements in an integer list. To measure the amount of work required, we could count the integer addition operations. For a list of 100 elements, there are 99 addition operations. Note, however, that we do not actually have to count the number of addition operations; it is some *function* of the number of elements (N) on the list. Therefore, we can express the number of addition operations in terms of N: For a list of N elements, there are $N - 1$ addition operations. Now we can compare the algorithms for the general case, not just for a specific list size.

Sometimes an operation so dominates an algorithm that the other operations fade into the background "noise." If we want to buy elephants and goldfish, for example, and we are considering two pet suppliers, we only need to compare the prices of elephants;

the cost of the goldfish is trivial in comparison. Suppose we have two files of integers, and we want to create a new file of integers based on the sums of pairs of integers from the existing files. In analyzing an algorithm that solves this problem, we could count both file accesses and integer additions. However, file accessing is so much more expensive than integer addition in terms of computer time, that the integer additions could be a trivial factor in the efficiency of the whole algorithm; we might as well count only the file accesses, ignoring the integer additions. In analyzing algorithms, we often can find one operation that dominates the algorithm, effectively relegating the others to the "noise" level.



Big-O

We have been talking about work as a function of the size of the input to the operation (for instance, the number of elements on the list to be summed). We can express an approximation of this function using a mathematical notation called order of magnitude, or Big-O notation. (This is a letter O, not a zero.) The order of magnitude of a function is identified with the term in the function that increases fastest relative to the size of the problem. For instance, if

Big-O notation A notation that expresses computing time (complexity) as the term in a function that increases most rapidly relative to the size of a problem

$$f(N) = N^4 + 100N^2 + 10N + 50$$

then $f(N)$ is of order N^4—or, in Big-O notation, $O(N^4)$. That is, for large values of N, some multiple of N^4 dominates the function for sufficiently large values of N.

How is it that we can just drop the low-order terms? Remember the elephants and goldfish that we talked about earlier? The price of the elephants was so much greater that we could just ignore the price of the goldfish. Similarly, for large values of N, N^4 is so much larger than 50, $10N$, or even $100N2$ that we can ignore these other terms. This doesn't mean that the other terms do not contribute to the computing time; it only means that they are not significant in our approximation when N is "large."

What is this value N? N represents the size of the problem. Most of the rest of the problems in this book involve data structures—lists, stacks, queues, and trees. Each structure is composed of elements. We develop algorithms to add an element to the structure and to modify or delete an element from the structure. We can describe the work done by these operations in terms of N, where N is the number of elements in the structure. Yes, we know. We have called the number of elements in a list the length of the list. However, mathematicians talk in terms of *N*, so we use *N* for the length when we are comparing algorithms using Big-O notation.

Suppose that we want to write all the elements in a list into a file. How much work is that? The answer depends on how many elements are on the list. Our algorithm is

Write List Elements

```
Open the file
while more elements in list
    Write the next element
```

If *N* is the number of elements on the list, the “time” required to do this task is

*(N * time-to-write-one-element) + time-to-open-the-file*

This algorithm is O(*N*) because the time required to perform the task is proportional to the number of elements (*N*)—plus a little to open the file. How can we ignore the open time in determining the Big-O approximation? If we assume that the time necessary to open a file is constant, this part of the algorithm is our goldfish. If the list has only a few elements, the time needed to open the file may seem significant, but for large values of *N*, writing the elements is an elephant in comparison with opening the file.

The order of magnitude of an algorithm does not tell you how long in microseconds the solution takes to run on your computer. Sometimes we need that kind of information. For instance, a word processor’s requirements state that the program must be able to spell-check a 50-page document (on a particular computer) in less than 120 seconds. For information like this, we do not use Big-O analysis; we use other measurements. We can compare different implementations of a data structure by coding them and then running a test, recording the time on the computer’s clock before and after. This kind of “benchmark” test tells us how long the operations take on a particular computer, using a particular compiler. The Big-O analysis, however, allows us to compare algorithms without reference to these factors.

Common Orders of Magnitude

O(1) is called bounded time. The amount of work is bounded by a constant and is not dependent on the size of the problem. Assigning a value to the *i*th element in an array of *N* elements is O(l) because an element in an array can be accessed directly through its index. Although bounded time is often called constant time, the amount of work is not necessarily constant. It is, however, bounded by a constant.

O(log2N) is called *logarithmic* time. The amount of work depends on the log of the size of the problem. Algorithms that successively cut the amount of data to be processed in half at each step typically fall into this category. Finding a value in a list of sorted elements using the binary search algorithm is O($\log_2 N$).

O(N) is called *linear* time. The amount of work is some constant times the size of the problem. Printing all the elements in a list of N elements is O(N). Searching for a particular value in a list of unsorted elements is also O(N) because you must potentially search every element on the list to find it.

O($N \log_2 N$) is called (for lack of a better term) $N \log_2 N$ time. Algorithms of this type typically involve applying a logarithmic algorithm N times. The better sorting algorithms, such as Quicksort, Heapsort, and Mergesort discussed in Chapter 10, have $N \log_2 N$ complexity. That is, these algorithms can transform an unsorted list into a sorted list in O($N \log_2 N$) time.

O(N^2) is called *quadratic* time. Algorithms of this type typically involve applying a linear algorithm N times. Most simple sorting algorithms are O(N^2) algorithms. (See Chapter 10.)

O(2^N) is called *exponential* time. These algorithms are extremely costly. An example of a problem for which the best known solution is exponential is the traveling salesman problem—given a set of cities and a set of roads that connect some of them, plus the lengths of the roads, find a route that visits every city exactly once and minimizes total travel distance. As you can see in Table 3.3, exponential times increase dramatically in relation to the size of N. (It also is interesting to note that the values in the last column grow so quickly that the computation time required for problems of this order may exceed the estimated life span of the universe!)

Note that throughout this discussion we have been talking about the amount of work the computer must do to execute an algorithm. This determination does not necessarily relate to the size of the algorithm, say, in lines of code. Consider the following two algorithms to initialize to zero every element in an N-element array.

Algorithm Init1

```
items[0] = 0;
items[1] = 0;
items[2] = 0;
items[3] = 0;
.
.
.
items[N-1] = 0;
```

Algorithm Init2

```
for (index = 0; index < N; index++)
  items[index] = 0;
```

Both algorithms are O(N), even though they greatly differ in the number of lines of code.

Table 3.3 *Comparison of rates of growth*

N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4,096	65,536
32	5	160	1,024	32,768	4,294,967,296
64	6	384	4,096	262,144	About 5 years' worth of instructions on a supercomputer
128	7	896	16,384	2,097,152	About 600,000 times greater than the age of the universe in nanoseconds (for a 6-billion-year estimate)
256	8	2,048	65,536	16,777,216	Don't ask!

Now let's look at two different algorithms that calculate the sum of the integers from 1 to *N*. Algorithm Sum1 is a simple *for* loop that adds successive integers to keep a running total:

Algorithm Sum1

```
sum = 0;
for (count = 1; count <= n; count++)
   sum = sum + count;
```

That seems simple enough. The second algorithm calculates the sum by using a formula. To understand the formula, consider the following calculation when $N = 9$.

```
  1 +  2 +  3 +  4 +  5 +  6 +  7 +  8 +  9
+ 9 +  8 +  7 +  6 +  5 +  4 +  3 +  2 +  1
-----------------------------------------------
 10 + 10 + 10 + 10 + 10 + 10 + 10 + 10 + 10   =   10 * 9 = 90
```

We pair up each number from 1 to N with another, such that each pair adds up to $N + 1$. There are N such pairs, giving us a total of $(N + 1)*N$. Now, because each number is included twice, we divide the product by 2. Using this formula, we can solve the problem: ((9 + 1) * 9)/2 = 45. Now we have a second algorithm:

Algorithm Sum2

```
sum = ((n + 1) * n) / 2;
```

Both of the algorithms are short pieces of code. Let's compare them using Big-O notation. The work done by Sum1 is a function of the magnitude of N; as N gets larger, the amount of work grows proportionally. If N is 50, Sum1 works 10 times as hard as when N is 5. Algorithm Sum1, therefore, is $O(N)$.

To analyze Sum2, consider the cases when $N = 5$ and $N = 50$. They should take the same amount of time. In fact, whatever value we assign to N, the algorithm does the same amount of work to solve the problem. Algorithm Sum2, therefore, is $O(1)$.

Does this mean that Sum2 is always faster? Is it always a better choice than Sum1? That depends. Sum2 might seem to do more "work," because the formula involves multiplication and division, whereas Sum1 is a simple running total. In fact, for very small values of N, Sum2 actually might do more work than Sum1. (Of course, for very large values of N, Sum1 does a proportionally larger amount of work, whereas Sum2 stays the same.) So the choice between the algorithms depends in part on how they are used, for small or large values of N.

Another issue is the fact that Sum2 is not as obvious as Sum1, and thus it is harder for the programmer (a human) to understand. Sometimes a more efficient solution to a problem is more complicated; we may save computer time at the expense of the programmer's time.

So, what's the verdict? As usual in the design of computer programs, there are tradeoffs. We must look at our program's requirements and then decide which solution is better. Throughout this text we examine different choices of algorithms and data structures. We compare them using Big-O, but we also examine the program's requirements and the "elegance" of the competing solutions. As programmers, we design software solutions with many factors in mind.

Family Laundry: An Analogy

How long does it take to do a family's weekly laundry? We might describe the answer to this question with the function

$$f(N) = c * N$$

where N represents the number of family members and c is the average number of minutes that each person's laundry takes. We say that this function is O(N) because the total laundry time depends on the number of people in the family. The "constant" c may vary a little for different families—depending on the size of their washing machine and how fast they can fold clothes, for instance. That is, the time to do the laundry for two different families might be represented with these functions:

$$f(N) = 100 * N$$
$$g(N) = 90 * N$$

But overall, we describe these functions as O(N).

Now what happens if Grandma and Grandpa come to visit the first family for a week or two? The laundry time function becomes

$$f(N) = 100 * (N + 2)$$

We still say that the function is O(N). How can that be? Doesn't the laundry for two extra people take any time to wash, dry, and fold? Of course it does! If N is small (the family consists of Mother, Father, and Baby Sierra), the extra laundry for two people is significant. But as N grows large (the family consists of Mother, Father, 8 kids, and a dog named Waldo), the extra laundry for two people doesn't make much difference. (The family's laundry is the elephant; the guest's laundry is the goldfish.) When we compare algorithms using Big-O, we are concerned with what happens when N is "large."

If we are asking the question "Can we finish the laundry in time to make the 7:05 train?" we want a precise answer. The Big-O analysis doesn't give us this information. It gives us an approximation. So, if 100 * N, 90 * N, and 100 * (N + 2) are all O(N), how can we say which is better? We can't—in Big-O terms, they are all roughly equivalent for large values of N. Can we find a better algorithm for getting the laundry done? If the family wins the state lottery, they can drop all their dirty clothes at a professional laundry 15 minutes' drive from their house (30 minutes round trip). Now the function is

$$f(N) = 30$$

This function is O(1). The answer is not dependent on the number of people in the family. If they switch to a laundry 5 minutes from their house, the function becomes

$$f(N) = 10$$

This function is also O(1). In terms of Big-O, the two professional-laundry solutions are equivalent: No matter how many family members or houseguests you have, it takes a constant amount of the family's time to do the laundry. (We aren't concerned with the professional laundry's time.)

3.6 Comparison of Unsorted and Sorted List ADT Algorithms

In order to determine the Big-O notation for the complexity of these algorithms, we must first determine the size factor. Here we are considering algorithms to manipulate items in a list. Therefore, the size factor is the number of items on the list: `numItems`.

Many of our algorithms are identical for the Unsorted List ADT and the Sorted List ADT. We capitalized on this fact in Section 3.4 when we brought the corresponding methods together in our abstract list class. Let's examine these first. The `lengthIs` and `isFull` methods each contain only one statement: `return numItems` and `return (list.length == numItems)`. Since the number of statements executed in these methods does not depend on the number of items on the list, they have O(1) complexity. The `reset` method contains one assignment statement and `getNextItem` contains an assignment statement, an *if-then-else* statement, and a *return* statement. Neither of these methods is dependent on the number of items on the list, so they also have O(1) complexity. The other methods are different for the two implementations.

Unsorted List ADT

The algorithm for `isThere` requires that the list be searched until an item is found or the end of the list is reached. We might find the item in any position on the list, or we might not find it at all. How many places must we examine? At best only one, at worst `numItems`. If we took the best case as our measure of complexity, then all of the operations would have O(1) complexity. But this is a rare case. What we want is the average case or worst case, which in this instance are the same: O(`numItems`). True, the average case would be O(`numItems/2`), but when we are using order notation, O(`numItems`) and O(`numItems/2`) are equivalent. In some cases that we discuss later, the average and the worst cases are not the same.

The `insert` algorithm has two parts: find the place to insert the item and insert the item. In the unsorted list, the item is put in the `numItems` position and `numItems` is incremented. Neither of these operations is dependent on the number of items on the list, so the complexity is O(1).

The `delete` algorithm has two parts: find the item to delete and delete the item. Finding the item uses essentially the same algorithm as `isThere`. The only difference is that since it is guaranteed that the item is on the list, we do not have to test for the end-of-list condition. But that difference does not affect the number of times we may have to traverse the search loop, so the complexity of that part is O(`numItems`). To delete the item, we put the value in the `numItems - 1` position into the location of the item to be deleted and decrement `numItems`. This store and decrement are not dependent on the number of items on the list, so this part of the operation has complexity O(1). The entire delete algorithm has complexity O(`numItems`) because O(`numItems`) plus O(1) is O(`numItems`). (Remember, the O(1) is the goldfish.)

Sorted List ADT

We looked at three different algorithms for `isThere`. We said that the Unsorted List ADT algorithm would work for a sorted list but that there were two more efficient algorithms: a linear search in the sorted list that exits when the place where the item would be is passed and a binary search.

A linear search in a sorted list is faster than in an unsorted list when searching for an item that is not on the list, but is the same when searching for an item that is on the list. Therefore, the complexity of the linear search in a sorted list is the same as the complexity in an unsorted list: O(`numItems`). Does that mean that we shouldn't bother taking advantage of the ordering in our search? No, it just means that the Big-O complexity measures are the same.

What about the binary search algorithm? We showed a table comparing the number of items searched in a linear search versus a binary search for certain sizes of lists. How do we describe this algorithm using Big-O notation? To figure this out, let's see how many times we can split a list of N items in half. Assuming that we don't find the item we are looking for at one of the earlier midpoints, we have to divide the list $\log_2 N$ times at the most, before we run out of elements to split. In case you aren't familiar with logs,

$$2^{log_2 N} = N$$

The definition of $\log_2 N$ is "the number that you raise 2 to, to get N". So, if we raise 2 to that number, $2^{\log_2 N}$, the result is N. Consider, for example, that if $N = 1024$, $\log_2 N = 10$, and $2^{10} = 1024$. How does that apply to our searching algorithms? The sequential search is $O(N)$; in the worst case, we would have to search all 1024 elements of the list. The binary search is $O(\log_2 N)$; in the worst case we would have to make $\log_2 N + 1$, or 11, search comparisons. A heuristic (a rule of thumb) tells us that a problem that is solved by successively splitting it in half is an $O(\log_2 N)$ algorithm. Figure 3.12 illustrates the relative growth of the linear and binary searches, measured in number of comparisons.

The `insert` algorithm still has the same two parts: finding the place to insert the item and inserting the item. Because the list must remain sorted, we must search for the position into which the new item must go. Our algorithm used a linear search to find

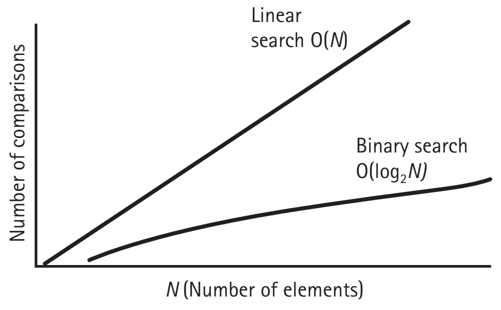


Figure 3.12 *Comparison of linear and binary searches*

the appropriate location: O(`numItems`). Inserting requires that we move all those elements from the insertion point down one place in the array. How many items must we move? At most `numItems`, giving us O(`numItems`). O(`numItems`) plus O(`numItems`) is O(`numItems`) because we disregard the constant 2. Note, however, that the constant 2 does not actually occur here. We actually access each item on the list only once except for the item at the insertion point: We access those to the place of insertion and we move those items stored from `numItems - 1` through that place. Therefore, only the element in the insertion location is accessed twice: once to find the insertion point and once to move it.

You may have thought of an even more efficient way to insert the item. You could start at the end of the list and repeatedly test to see if that is where you need to put the item. If the item is larger then the element at the end of the list, you just insert it following that element; if it is not you move the list element at the end of the list down one array position, and check the next to last list element, repeating the same pattern of compare and move. By the time you find out where to insert the item, you have already shifted all of the elements that are greater than down one location in the array, and you can just insert it into the open location. With this approach, on average, you only have to access half of the elements in the array, instead of all of the elements. However, it is still the same complexity as the other approach, since O(`numItems/2`) is equal to O(`numItems`).

The `delete` algorithm also still has the same two parts: finding the item to delete and deleting the item. The algorithm for finding the item is the mirror image of finding the insertion point: O(`numItems`). Deleting the item in a sorted list requires that all the elements from the deletion location to the end of the list must be moved forward one position. This shifting algorithm is the reverse of the shifting algorithm in the insertion and, therefore, has the same complexity: O(`numItems`). Hence the complexities of the insertion and deletion algorithms are the same in the Sorted List ADT.

Table 3.4 summarizes these complexities. We have replaced `numItems` with N, the generic name for the size factor.

In the deletion operation, we could improve the efficiency by using the binary search algorithm to find the item to delete. Would this change the complexity? No, it would not. The find would be $O(\log_2 N)$, but the removal would still be $O(N)$; since $O(\log_2 N)$ combined with $O(N)$ is $O(N)$ we have not changed the overall complexity of the algorithm. (Recall that the term with the largest power of N dominates.) Does this mean that we should not use the binary search algorithm? No, it just means that as the length of the list grows, the cost of the removal dominates the cost of the find.

Think of the common orders of complexity as being bins into which we sort algorithms (Figure 3.13). For small values of the size factor, an algorithm in one bin may actually be faster than the equivalent algorithm in the next-more-efficient bin. As the size factor increases, the differences among algorithms in the different bins get larger. When choosing between algorithms within the same bin, you look at the constants to determine which to use.

Table 3.4 *Big-O comparison of list operations*

Operation	Unsorted List	Sorted List
`length`	O(1)	O(1)
`isFull`	O(1)	O(1)
`reset`	O(1)	O(1)
`getNextItem`	O(1)	O(1)
`isThere`	O(*N*)	O(*N*)
		O($\log_2 N$) binary search
`insert`		
Find	O(1)	O(*N*)
Put	O(1)	O(*N*)
Combined	O(1)	O(*N*)
`delete`		
Find	O(*N*)	O(*N*)
Put	O(1)	O(*N*)
Combined	O(*N*)	O(*N*)

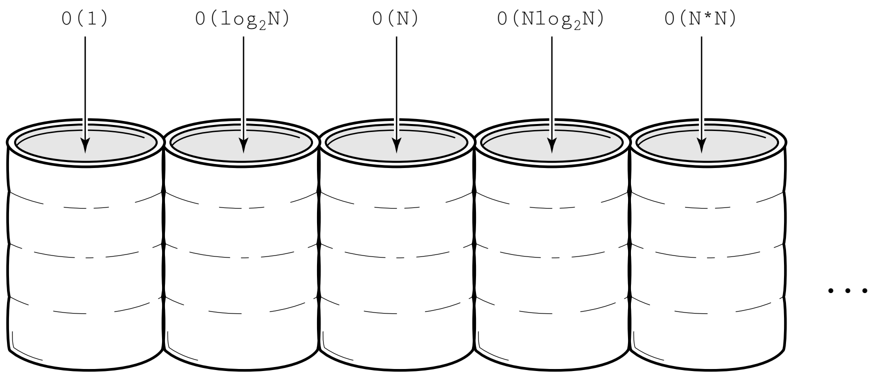


Figure 3.13 *Complexity bins*

3.7 Generic ADTs

So far in this chapter we have created several variations of list ADTs: a "standalone" unsorted string list, an unsorted string list that extended an abstract list class, and a sorted string list, that also extended the abstract list class. These string lists are very useful to an application programmer who is creating a system that requires lists of strings. But what if the programmer wanted some other kind of list: a list of integers, a list of dates, a list of circles, a list of real estate information?

The list ADTs we have constructed so far have all been constrained to holding data of one specific type, namely strings. While useful, think of how much more useful they would be if they could hold any kind of information. A generic data type is one for which the operations are defined but the types of the items being manipulated are not. We can make our lists generic by using Java's *interface* construct. We limited ourselves to lists of strings up until now, because we wanted to concentrate on the list operations without dealing with the extra complexity of interfaces. Now, however, we are ready to see how we can construct more generally usable ADTs.

Generic data type A type for which the operations are defined but the types of the items being manipulated are not

We use a new package, `ch03.genericLists`, to organize our files related to generic lists. As required, the files are placed in a subdirectory `genericLists` of the subdirectory `ch03` of the directory `bookFiles`. Additionally, each of the class files must begin with the line

```
package ch03.genericLists;
```

Lists of Objects

One approach to creating generic ADTs is to have our ADTs use variables of type `Object`. Since all Java classes ultimately inherit from `Object`, such an ADT should be able to "hold" a variable of any class. If you try this approach, you soon see that it has severe limitations.

Consider what happens if you redefine our `SortedStringList` class to hold objects instead of strings. If you edit the file containing the class, and change every place where you see "String" with "Object", you have created a `SortedObjectList` class. At first glance this seems to have solved our problem. The list is implemented as an array of objects. We can insert objects into the list and delete them. Many of the methods, like `isFull` and `reset`, are not even affected by the change. However, when you try to compile the file you discover a few errors. For example, the following line from the `insert` method of the new file is flagged with a "method not found" message:

```
if (item.compareTo(list[location]) < 0)
```

Do you see why? Remember that the `item` referred to in the code is now of class `Object`. If you check the definition of the `Object` class you see that it does not include a `compareTo` method. Therefore, this statement, along with several other statements in the file, is syntactically illegal. The statement was OK when `item` was a string, since the `String` class includes a `compareTo` method, but it is not a legal statement when `item` is an object of the more general `Object` class.

The `String` class's `compareTo` method returns information about the relative ordering of two strings. Such a method is not defined for the `Object` class, since it might not always make sense to talk about the ordering of two objects. We cannot have a sorted list of just any type of objects. We can only have a sorted list of objects for which a relative ordering has been defined.

There is one other kind of statement that is flagged by the compiler. This statement also appears in the `insert` method:

```
list[location] = new Object(item);
```

You should recall that this statement is executed after the method has shifted the array values to make room for the new item and set the value of `location` to the insertion location. The previous form of this statement used the `String` class's copy constructor, `String(item)`, to create a new string object, which was then inserted into the list. The new form of this statement attempts to use a copy constructor from the `Object` class, but no such constructor exists. The reason we wish to insert a copy of the item into the list, instead of just inserting the item itself, is to preserve the information hiding aspect of our ADT.

To solve these problems we create a Java interface with abstract classes for comparing and copying objects.

The Listable Interface

To ensure that the objects that we place on our list support the necessary methods, we create a Java interface. Recall from Chapter 2 that an interface can only include abstract methods, that is, methods without bodies. Once the interface is defined we can create classes that implement the interface by supplying the missing method bodies.

For our lists we create an interface with two abstract methods; one to compare elements so that we can support sorted lists and the `isThere` operation, and one to support copying of list elements, so that we can maintain information hiding. We follow the Java convention used in the `String` class by naming the former method `compareTo` and by having it return integer values to indicate the result of the comparison. We call the latter method `copy`. It does not need any parameters; it simply returns a copy of the object on which it is invoked. Finally, we need a name for the interface itself. Let's call it `Listable`, since classes that implement this interface provide objects that can be listed.

Here is the code for the interface:

```
package ch03.genericLists;

public interface Listable
// Objects of classes that implement this interface can be used with lists
```

```
{
  public abstract int compareTo(Listable other);
  // Compares this Listable object to "other". If they are equal, 0 is
  //   returned
  // If this is less than the argument, a negative value is returned
  // If this is more than the argument, a positive value is returned

  public abstract Listable copy();
  // Returns a new object with the same contents as this Listable object
}
```

Whatever data we intend to store on a list must be contained in a class that implements the `Listable` interface. For example, to support a list of circles we might define a `ListCircle` class as follows (some of the code that is not pertinent to this discussion has been left out):

```
package ch03.genericLists;

public class ListCircle implements Listable
{
  private int xvalue;       // Horizontal position of center
  private int yvalue;       // Vertical position of center
  private float radius;
  private boolean solid;    // True means circle filled

  // Code for Constructors goes here

  public int compareTo(Listable otherCircle)
  {
    ListCircle other = (ListCircle)otherCircle;
    return (int)(this.radius - other.radius);
  }

  public Listable copy()
  {
    ListCircle result = new ListCircle(this.xvalue, this.yvalue, this.radius,
                                       this.solid);
    return result;
  }

  // More ListCircle methods as needed

}
```

Note the use of the cast operation (ListCircle) in the compareTo method:

```
ListCircle other = (ListCircle)otherCircle;
```

This is to ensure that the parameter otherCircle is a ListCircle. The method signature allows it to be any Listable type, yet on the following line we are assuming that it is a ListCircle, when we access its radius instance variable.

Since ListCircle implements Listable, it can be used anywhere something of type Listable is expected. In the next section we define a class that provides a list of Listable objects. This class could therefore be used to provide a list of ListCircle objects.

A Generic Abstract List Class

Now we can create our generic list ADT by defining a list of Listable elements; not just strings, not just plain objects, but objects of classes that implement the Listable interface. We can reuse the code from our previous list definitions, but we must replace the use of the String class with the Listable interface throughout the code; we also must replace the use of the String class's copy constructor with statements that use the copy method defined in the interface.

We no longer need to use the term "string" when defining our list classes, since they are no longer constrained to providing only lists of strings. We call our new abstract list class simply List. Below is the code for the abstract List class. There are several things to notice about the code. First, note the use of the term Listable, in place of a class or type name, throughout the code. Wherever Listable is used to represent a formal parameter, you can pass an object of a class that implements Listable, as the actual parameter. For example, you could use objects of type ListCircle, which was defined in the previous subsection. Alternately, if you have defined other classes that implement the Listable interface, you could use objects of those classes—perhaps a class of ListStrings or a class of ListStudents.

Also, note the invocation of the copy method on the next object, in the very last statement of the class. The next object is of "type" Listable, that is, it is an object of a class that implements Listable. Therefore, we can be assured that the creator of that class has included a definition of the copy method within the class.

Finally, you should notice the addition of a new list method, retrieve, and some small but important changes to the comments describing the effects of the methods isThere and delete. The switch from supporting lists of strings to lists of Listable objects means that we now can implement and use lists of composite elements. This raises some interesting questions about how we compare elements, and what it means for two elements to be "equal." These questions are discussed following the code listing.

```
//----------------------------------------------------------------------------
// List.java                by Dale/Joyce/Weems                      Chapter 3
//
// Defines all constructs for an array based list that do not depend
// on whether or not the list is sorted.
//----------------------------------------------------------------------------

package ch03.genericLists;

public abstract class List
{
  protected Listable[] list;          // Array to hold this list's elements
  protected int numItems;             // Number of elements on this list
  protected int currentPos;           // Current position for iteration

  public List(int maxItems)
  // Instantiates and returns a reference to an empty list object
  // with room for maxItems elements
  {
    numItems = 0;
    list = new Listable[maxItems];
  }

  public boolean isFull()
  // Returns whether this list is full
  {
    return (list.length == numItems);
  }

  public int lengthIs()
  // Returns the number of elements on this list
  {
    return numItems;
  }

  public abstract boolean isThere (Listable item);
  // Returns true if an element with the same key as item is on this list;
  // otherwise, returns false

  public abstract Listable retrieve(Listable item);
  // Returns a copy of the list element with the same key as item

  public abstract void insert (Listable item);
  // Adds a copy of item to this list

  public abstract void delete (Listable item);
  // Deletes the element with the same key as item from this list

  public void reset()
  // Initializes current position for an iteration through this list
```

```
    {
      currentPos  = 0;
    }

    public Listable getNextItem ()
    // Returns copy of the next element on this list
    {
      Listable next = list[currentPos];
      if (currentPos == numItems-1)
        currentPos = 0;
      else
        currentPos++;
      return next.copy();
    }
  }
```

As mentioned above, the switch from supporting lists of strings to lists of `Listable` objects means that we now can implement and use lists of composite elements. This affects how we can compare elements, and what it means for two elements to be "equal." Consider the following `ListCircle` objects `C1`, `C2`, `C3`, and `C4`:

	C1	C2	C3	C4
xvalue	3	3	6	3
yvalue	4	4	12	4
radius	10	6	10	3
solid	true	false	false	true

Are any of the circles equal to each other? No, not in the strict sense of the word "equal." But what about equality as defined by the `compareTo` method of the `ListCircle` class? There, `ListCircle` objects are compared strictly on the basis of their radii. Based on that definition of equality, circles `C1` and `C3` are "equal." Although it might seem strange, this definition of equality could make perfect sense for a particular application, where the only important criteria for comparing circles is their size.

Remember that we are following the convention that our lists consist of unique objects. Are all of the circles in the table above unique? No, not in the "world" defined by the `ListCircle` class, where two circles are considered identical if they have the same `radius`. The `compareTo` method essentially defines the key for the list. In this case, the key is the radius. We should not insert both `C1` and `C3` on the same list. That would violate the precondition of the `insert` operation. Again, this seems like a strange restriction but might make sense within a particular application. (Please remember that the approach used here is not the only approach possible. For example, list ADTs could be developed that separate the concepts of key values and sort values.)

Let's look at another example. Earlier in this chapter we discussed different ways we might wish to sort a list of student records, with each record containing fields for first name, last name, identification number, and three test scores. For example, we could sort the list by name, or we could sort the list by identification number. Here is a table of values for student objects S1, S2, and S3.

	S1	S2	S3
first	Jones	Jones	Adams
last	David	Mary	Mark
IDnum	1234567	7654321	1111111
test1	89	92	100
test2	92	95	99
test3	95	89	100

In our approach, the field or fields that we use as a sorting criteria is the key for the list. If we were to define a ListStudent class—a class that allows us to maintain a list of students—then the definition of the compareTo operation in the ListStudent class would effectively define the key for the list elements. What would be the best choice for the key for a list of students? If we decide the last name is the key, then we are not able to hold both S1 and S2 on our list. That does not seem reasonable. Perhaps we could define the key to use the first name field as a tiebreaker when two last names are identical. That is better, but we could have two students who have identical first and last names, in which case we would be in trouble again. For this information, assuming a unique identification number has been assigned to each student, the IDnum field would be the best key. Therefore, the compareTo method of the ListStudent class should base its processing on a comparison of IDnum values.

Now it is clear why we changed the comment describing the effects of the isThere and delete operations. For example, when we were just using a list of strings the effect of isThere was "returns true if item is on the list . . . ." Now the effect is "returns true if an element with the same key as item is on this list . . . ." When dealing with lists of noncomposite elements, like strings, the entire element was in effect the key. That is no longer the case.

This brings us to the new list operation introduced in this section, the retrieve operation. Its definition in the abstract List class is

```
public abstract Listable retrieve(Listable item);
// Returns a copy of the list element with the same key as item
```

The application passes retrieve a Listable object and retrieve searches the list to find the element on the list that is "equal" (i.e., has the same key) to it. A copy of this element is returned. The specification of retrieve is as follows.

Listable retrieve (Listable item)

Effect:	Returns a copy of the list element with the same key as item.
Preconditions:	An element with a key that matches item's key is on this list.
Postcondition:	Return value = (copy of list element that matches item)

Therefore, we can store information on a list and retrieve it later based on the item's key. For example, to retrieve student information about a student with an IDnum of 7654321, we instantiate a ListStudent object with dummy information for all of the fields except the IDnum field, which we initialize to 7654321. Then we pass this object to the retrieve operation, which returns a copy of the matching list element. This copy contains all the valid information about the student.

A Generic Sorted List ADT

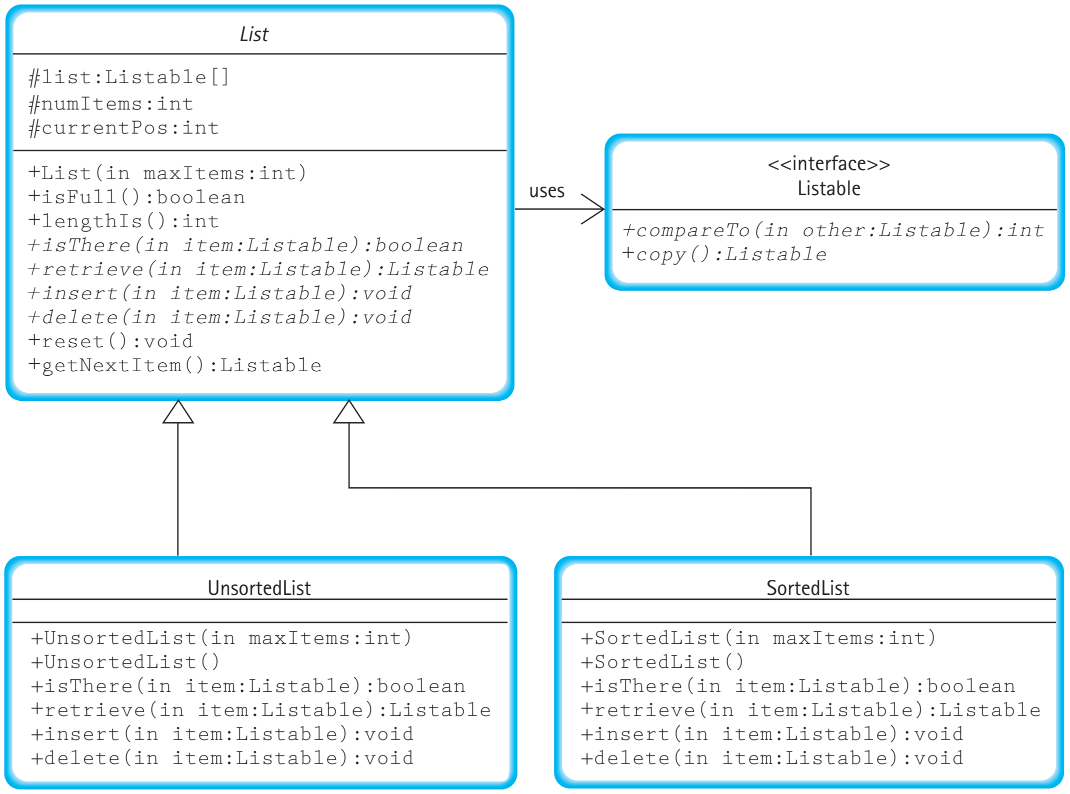
Next we list the code for the generic sorted list class that completes the definition of the list class for the case of sorted lists. You can see that we use the binary search algorithm to implement the isThere and retrieve operations (although with the retrieve operation we do not need the moreToSearch variable because we know the item being retrieved is on the list). Note the use of the term Listable throughout the class, the use of the copy method invocation in the insert and retrieve methods, and several uses of the compareTo method. We call our new sorted list class SortedList.

```
//----------------------------------------------------------------------------
// SortedList.java            by Dale/Joyce/Weems                   Chapter 3
//
// Completes the definition of the List class under the assumption
// that the list is kept sorted
//----------------------------------------------------------------------------

package ch03.genericLists;

public class SortedList extends List
{
  public SortedList(int maxItems)
  // Instantiates and returns a reference to an empty list object
  // with room for maxItems elements
  {
    super(maxItems);
  }
```

```
public SortedList()
// Instantiates and returns a reference to an empty list object
// with room for 100 elements
{
  super(100);
}

public boolean isThere (Listable item)
// Returns true if an element with the same key as item is on this list;
// otherwise, returns false
{
  int compareResult;
  int midPoint;
  int first = 0;
  int last = numItems - 1;
  boolean moreToSearch = (first <= last);
  boolean found = false;

  while (moreToSearch && !found)
  {
    midPoint = (first + last) / 2;
    compareResult = item.compareTo(list[midPoint]);

    if (compareResult == 0)
      found = true;
    else if (compareResult < 0)  // item is less than element at location
    {
      last = midPoint - 1;
      moreToSearch = (first <= last);
    }
    else                         // item is greater than element at location
    {
      first = midPoint + 1;
      moreToSearch = (first <= last);
    }
  }

 return found;
}

public Listable retrieve (Listable item)
// Returns a copy of the list element with the same key as item
{
  int compareResult;
  int first = 0;
  int last = numItems - 1;
  int midPoint = (first + last) / 2;
  boolean found = false;

  while (!found)
  {
    midPoint = (first + last) / 2;
    compareResult = item.compareTo(list[midPoint]);

    if (compareResult == 0)
      found = true;
    else if (compareResult < 0)  // item is less than element at location
      last = midPoint - 1;
    else                         // item is greater than element at location
      first = midPoint + 1;
  }

 return list[midPoint].copy();
}



public void insert (Listable item)
// Adds a copy of item to this list
{
  int location = 0;
  boolean moreToSearch = (location < numItems);

  while (moreToSearch)
  {
    if (item.compareTo(list[location]) < 0)  // item is less
      moreToSearch = false;
    else                                     // item is more
    {
      location++;
      moreToSearch = (location < numItems);
    }
  }

  for (int index = numItems; index > location; index--)
    list[index] = list[index - 1];

  list[location] = item.copy();
  numItems++;
}
```

```
  public void delete (Listable item)
  // Deletes the element that matches item from this list.
  {
    int location = 0;

    while (item.compareTo(list[location]) != 0)
      location++;

    for (int index = location + 1; index < numItems; index++)
      list[index - 1] = list[index];

    numItems--;
  }
}
```

The UML diagrams for the `List` and `SortedList` classes, plus the `Listable` interface, are displayed in Figure 3.14. Note the use of a dashed arrow labeled "uses," with an open arrowhead, to indicate the dependency of `List` and `Listable`. Although we



List
#list:Listable[]
#numItems:int
#currentPos:int
+List(in maxItems:int)
+isFull():boolean
+lengthIs():int
+isThere(in item:Listable):boolean
+retrieve(in item:Listable):Listable
+insert(in item:Listable):void
+delete(in item:Listable):void
+reset():void
+getNextItem():Listable
uses
<<interface>>
Listable
+compareTo(in other:Listable):int
+copy():Listable
UnsortedList
+UnsortedList(in maxItems:int)
+UnsortedList()
+isThere(in item:Listable):boolean
+retrieve(in item:Listable):Listable
+insert(in item:Listable):void
+delete(in item:Listable):void
SortedList
+SortedList(in maxItems:int)
+SortedList()
+isThere(in item:Listable):boolean
+retrieve(in item:Listable):Listable
+insert(in item:Listable):void
+delete(in item:Listable):void

Figure 3.14 *UML diagrams for our list framework*

did not develop it, the figure also shows an `UnsortedList` class that extends `List`. This helps remind us that more than one class can extend the abstract list class. The implementation of the `UnsortedList` class is left as an exercise.

A Listable Class

Now that we have defined a generic list, a sorted list of `Listable` elements, we have to define a class that implements the `Listable` interface so that we have something to put on our lists. To keep our example straightforward, we continue to work with a list of strings. (In the case study of the next section, we provide a more complicated example of a class that implements `Listable`.)

We used lists of strings in the early part of this chapter so that we could introduce the reader gently to the topic of defining and implementing ADTs in Java. Knowing what we know now, about how to use interfaces to create generic lists, we would not have created a specific list implementation for lists of strings. Instead, we would use our generic list. But how do we use our generic list to provide a list of strings? We need to create a new class that hides a string variable and implements the `Listable` interface. We call this class `ListString`, since it provides strings that can be placed on our generic list.

Study the code for `ListString` below. Note that it contains a single object variable `key` that holds a string. It provides a constructor, plus the two methods needed to implement the `Listable` interface, the `copy` and `compareTo` methods. It also contains one other method, a `toString` method, which makes it easy for the application programmer to use objects of the class `ListString` as strings. When a class implements an interface, it must provide concrete methods for the abstract methods defined in the interface. As you can see, it can also include definitions for other methods.

```
package ch03.genericLists;

public class ListString implements Listable
{
  private String key;

  public ListString(String inString)
  {
    key = new String(inString);
  }

  public Listable copy()
  {
    ListString result = new ListString(this.key);
    return result;
  }
  public int compareTo(Listable otherListString)
  {
   ListString other = (ListString)otherListString;
   return this.key.compareTo(other.key);
  }

  public String toString()
  {
    return(key);
  }
}
```

Since `ListString` implements `Listable`, objects of class `ListString` can be used anywhere a `Listable` object is expected. Therefore, an object of class `ListString` can be passed to the `insert` method of the `SortedList` class. Furthermore, the same object can be placed on the hidden array within the `SortedList` class. And so on. We can use `ListString` objects with the `SortedList` class to provide a sorted list of strings.

If we wished to have a list of something else we would need to create another class that implements `Listable`. For example, if we wished to have a list of `Circle` objects we could complete our definition of the class `ListCircle`. This class would also implement `Listable`; therefore, it would contain its own versions of the `copy` and `compareTo` methods. What does it mean to compare circles? That depends on the intended use of the list of circles. Perhaps the comparison would be based on the size of the circles or on their positions. The `ListCircle` class requires a constructor; but would it require a `toString` method? Would it require any other methods? Again, the answers depend on the intended use of the list of circles. Being able to reuse our generic list ADT with list elements that have been defined for a specific application provides us with a powerful programming tool.

Using the Generic List

To create a sorted list of strings in an application program you simply instantiate an object of the class `SortedList`, using either of its constructors:

```
SortedList list1 = new SortedList();
SortedList list2 = new SortedList(size);
```

You also need to declare at least one object of class `ListString`, so that you have a variable to use as a parameter with the various `SortedList` methods:

```
ListString aString;
```

Once these declarations have been made you can instantiate `ListString` objects and place them on the list. For example, to place the string "`Amy`" on the list you might code:

```
aString = new ListString("Amy");
list.insert(aString);
```

We are not going to list an entire application program that uses a sorted list of strings. We did create a test driver (a form of application) that you can study and use; it is in the `TDSortedList.java` file of the `ch03` subdirectory of the `bookFiles` directory on our website. Notice that it is not part of the `genericLists` package—instead it uses the package. So that the package classes are available to the test driver, it includes the following import statement:

```
import ch03.genericLists.*;
```

As long as the `bookFiles` directory is included on your computer's `ClassPath`, the compiler will know where to find the generic list files.

In the test driver you find uses of each of the sorted list methods with a `Listable` object:

```
outFile.println("The list is full is " + list.isFull());
outFile.println("Length of the list is " + list.lengthIs());
outFile.println(aString + " is on the list: " +
list.isThere(aString));
bString = (ListString)list.retrieve(aString);
list.insert(aString);
list.delete(aString);
list.reset();
aString = (ListString)list.getNextItem();
```

`SortedList.java` should be thoroughly tested. This job is left as an exercise.

The case study presented next shows another example of using the sorted list ADT. In the case study a list of real estate information is manipulated.

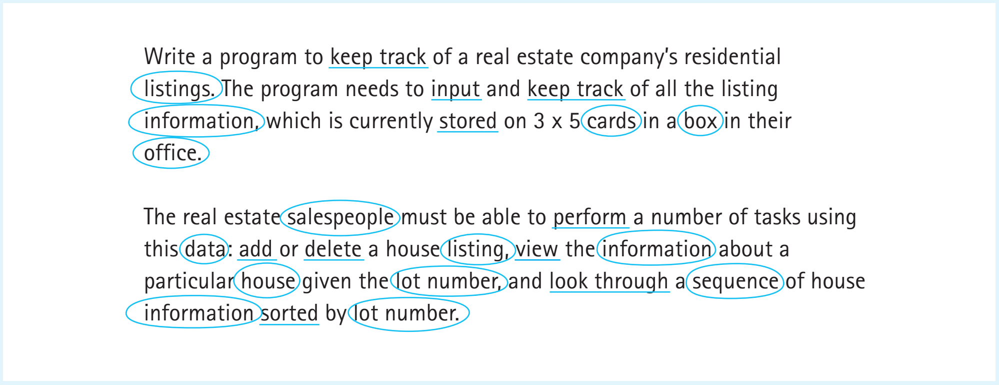
Case Study

Real Estate Listings

Problem Write a `RealEstate` program to keep track of a real estate company's residential listings. The program needs to input and keep track of all the listing information, which is currently stored on 3 × 5 cards in a box in their office.

The real estate salespeople must be able to perform a number of tasks using this data: add or delete a house listing, view the information about a particular house given the lot number, and look through a sequence of house information sorted by lot number.

We use the same design approach we described in Chapter 1 for this problem.



Write a program to keep track of a real estate company's residential listings. The program needs to input and keep track of all the listing information, which is currently stored on 3 x 5 cards in a box in their office.

The real estate salespeople must be able to perform a number of tasks using this data: add or delete a house listing, view the information about a particular house given the lot number, and look through a sequence of house information sorted by lot number.

Figure 3.15 *Problem statement with nouns circled and verbs underlined*

Brainstorming We said that nouns in the problem statement represent objects and that verbs describe actions. Let's approach this problem by analyzing the problem statement in terms of nouns and verbs. Let's circle nouns and underline verbs. The relevant nouns in the first paragraph are listings, information, cards, box, and office: circle them. The verbs that describe possible program actions are keep track, input, and stored: underline them. In the second paragraph, the nouns are salespeople, data, listing, information, house, lot number, and sequence: circle them. Possible action verbs are perform, add, delete, view, look through, and sorted: underline them. Figure 3.15 shows the problem statement with the nouns circled and the verbs underlined.

We did not circle program or underline write because these are instructions to the programmer and not part of the problem to be solved. Now, let's examine these nouns and verbs and see what insights they give us into the solution of this problem.

Filtering The first paragraph describes the current system. The objects are cards that contain information. These cards are stored in a box. Therefore, there are two objects in the office that we are going to have to simulate: 3×5 cards and a box to put them in. In the second paragraph, we discover several synonyms for the cards: data, listing, information, and house. We model these with the same objects that represent the cards. We also see what processing must be done with the cards and the box in which they are stored. The noun salespeople represents the outside world interacting with the program, so the rest of the paragraph describes the processing options that must be provided to the user of the program. In terms of the box of cards, the user must be able to add a new card, delete a card, view the information on the card given the lot number, and view a sequence of card information, sorted by lot number.

We can represent the cards by a class whose data members are the information written on the 3×5 cards. How do we represent the box of cards? We have just written several versions of the Abstract Data Type List. A list is a good candidate to simulate a box and the information on the list can be objects that represent the 3×5 cards. Since these objects represent the house information, and they should be kept on a list, let's call the class that models a card of house information `ListHouse`. We must make sure that our `ListHouse` class implements the `Listable` interface, since we wish to maintain a list of `ListHouse` objects.

We now know that our program uses a list of `ListHouse` objects. But which version of a list shall we use? The unsorted version or the sorted version? Because the user must be allowed to look through the "house information sorted by lot number," the sorted version is a better choice. In the `ListHouse` class we base the definition of the `compareTo` method on the house's lot number. This ensures that the houses are kept sorted by lot number, just the way we need them.

So far, we have ignored the noun office and the fact that the program should "input and keep track" of the cards. A box of cards is stored permanently in the office. A list is a structure that exists only as long as the program in which it is defined is running. But how do we keep track of the information between runs of the program? That is, how do we simulate the office in which the box resides? A file is the structure that is used for permanent storage of information. Hence, there are two representations of the box of cards. When the program is running, the box is represented as a list. When the program is not running, the box is represented as a file. The program must move the information on the cards from a file to a list as the first action, and from a list to a file as the last action. We relegate the responsibility of interacting with the file to a class called `HouseFile`.

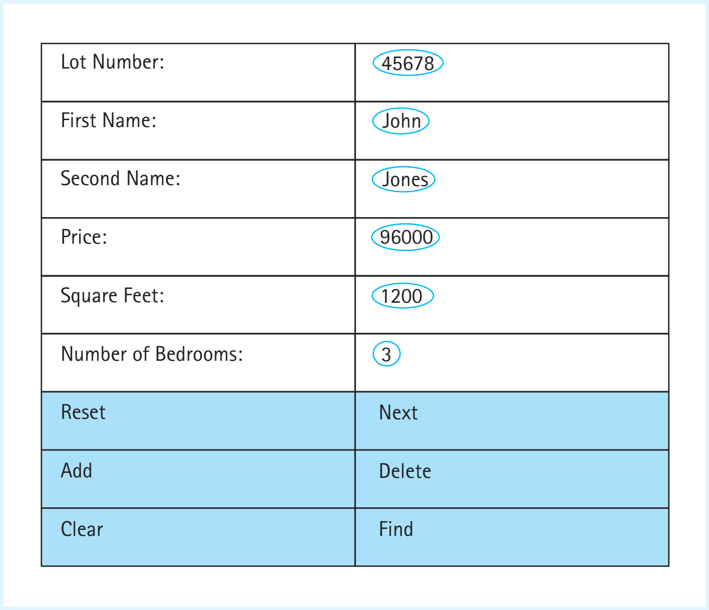
The `HouseFile` class hides the file of house information from the rest of the program. In this way, if the format of the file needs to be changed at a later time, the only part of the system that is affected is the `HouseFile` class. Limiting the scope of potential future changes is one of the main reasons we partition our systems into separate classes.

Let's capture the decisions we have made so far on CRC cards. On each card we record the main purpose of the class it represents, along with an initial set of responsibilities. Our cards show that our classes are already fairly well defined. The following table captures the information we record on the cards at this point (we display the final version of the cards after we finish the analysis section):

Class	Purpose	Responsibilities
`RealEstate`	Main program	Driver program, uses all the other classes to solve the problem; provides graphical user interface; implements actions represented by the interface buttons
`ListHouse`	Hold the information about a specific house.	Know all of its information; implement `Listable`; therefore, provide `copy` and `compareTo` methods
`SortedList`	Maintain a list of *ListHouse* elements.	See the List ADT specification.
`HouseFile`	Manage the file of house information.	Get house information from the file; save house information to the file.

User Interface Let's assume that the information on the 3 × 5 cards includes the owner's first and last names, the lot number, price, number of square feet, and number of bedrooms. The lot numbers are unique and therefore can be used as the key of the list. If an agent attempts to add a listing that duplicates an existing lot number, an error message is printed to the screen.

A review of the problem statement reveals that interaction with the user can take place one "house" at a time. Therefore, we design our graphical interface to display information about a house, and provide the user with buttons to initiate options related to that house (add, delete, clear) or to the overall system (reset, next, find). A count of the number of data fields, labels, and buttons needed, aided by some rough drafts drawn on scrap paper, leads us to a 9 × 2 grid layout for our interface. A sketch of our design is:



Lot Number:	45678
First Name:	John
Second Name:	Jones
Price:	96000
Square Feet:	1200
Number of Bedrooms:	3
Reset	Next
Add	Delete
Clear	Find

The user continues to manipulate the list of houses until he or she exits by closing the window.

Input Notice that there are three kinds of input: the file of houses saved from the last run of the program, the commands, and the data entered from the keyboard into the text fields in conjunction with the commands.

Output There are two kinds of output: the file of houses saved at the end of the run of the program, and screen output directed by one or more of the commands.

Data Objects There are house objects, represented in the program as `ListHouse` class objects. There are two container objects: the file of house objects retained from one run of the program to the next and the list into which the house objects are stored when the program is running (we call this object `list`). The collection of house listings is called our database.

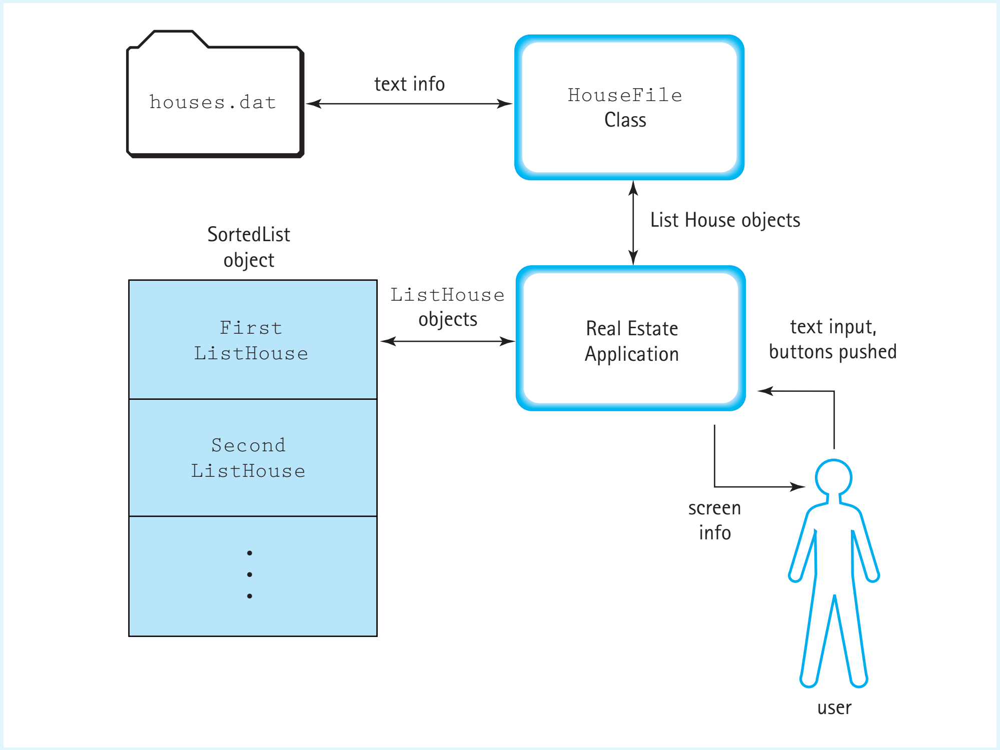


Figure 3.16 *Data flow of case study*

We name the physical file in which we retain the house objects `houses.dat`.

The diagrams in Figures 3.16 and 3.17 show the general flow of the processing and what the data objects look like. Note that we know the internal workings of the List ADT because we have just written the code earlier in the chapter. When we wrote that we were acting as the ADT programmer, creating a tool for use by application programmers. Now however, we are changing hats; we are acting as the application programmer. We write the program only using the interface as represented in the List ADT specification.

Scenario Analysis Where do we go from here? Scenario analysis lets us "test" our design. Using our CRC cards we can walk through several scenarios that represent the typical expected use of the system. This allows us to refine the responsibilities of our identified classes and begin to add detailed information about method names and interfaces. During the course of this analysis we may uncover holes in our identified classes or user interface.

We begin by working through a scenario in which a real estate salesperson runs the program and tries to get information about the house on lot number 45678. We realize that the first thing the system must do is to build the internal list of houses from the house information contained in the file. We need to decide which class should have this responsibility. We could

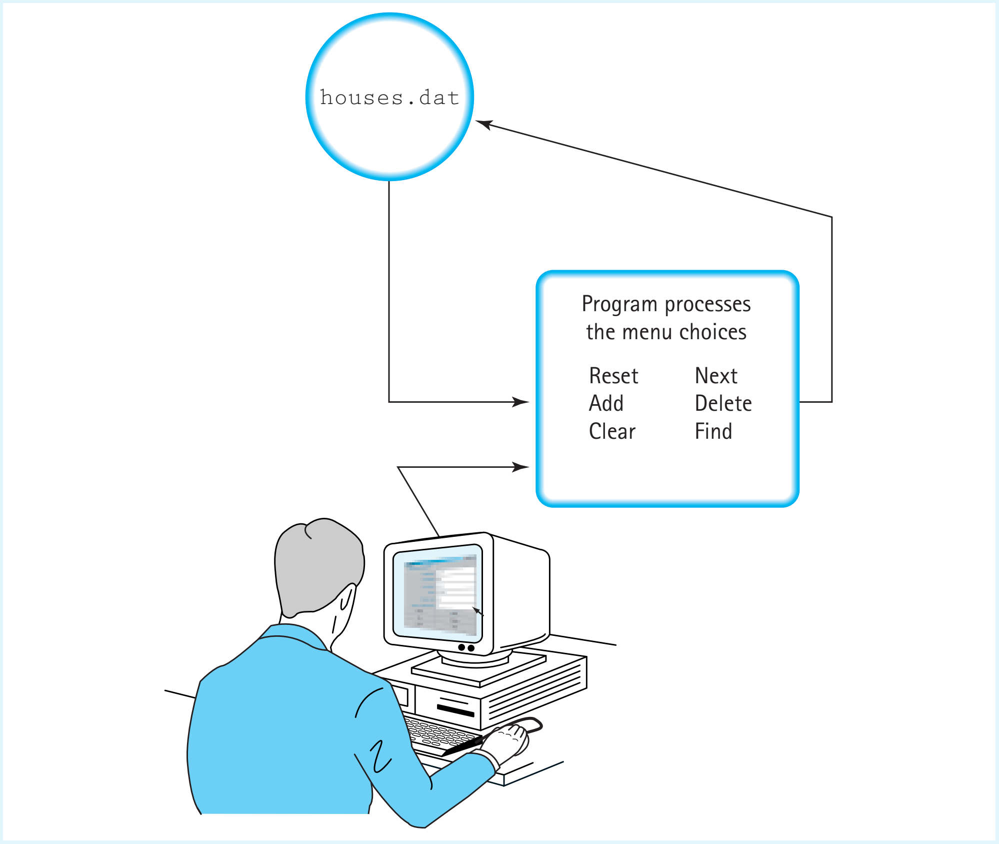


Figure 3.17 *The high-level processing of the case study*

assign this task to the `HouseFile` class, since it is able to get the house information from the file. However, we decide that such a task is outside its main purpose, which is to manage the file of house information. Therefore, we decide that the `RealEstate` class should perform this task. We add a notation to this effect to the list of responsibilities on its CRC card and move ahead.

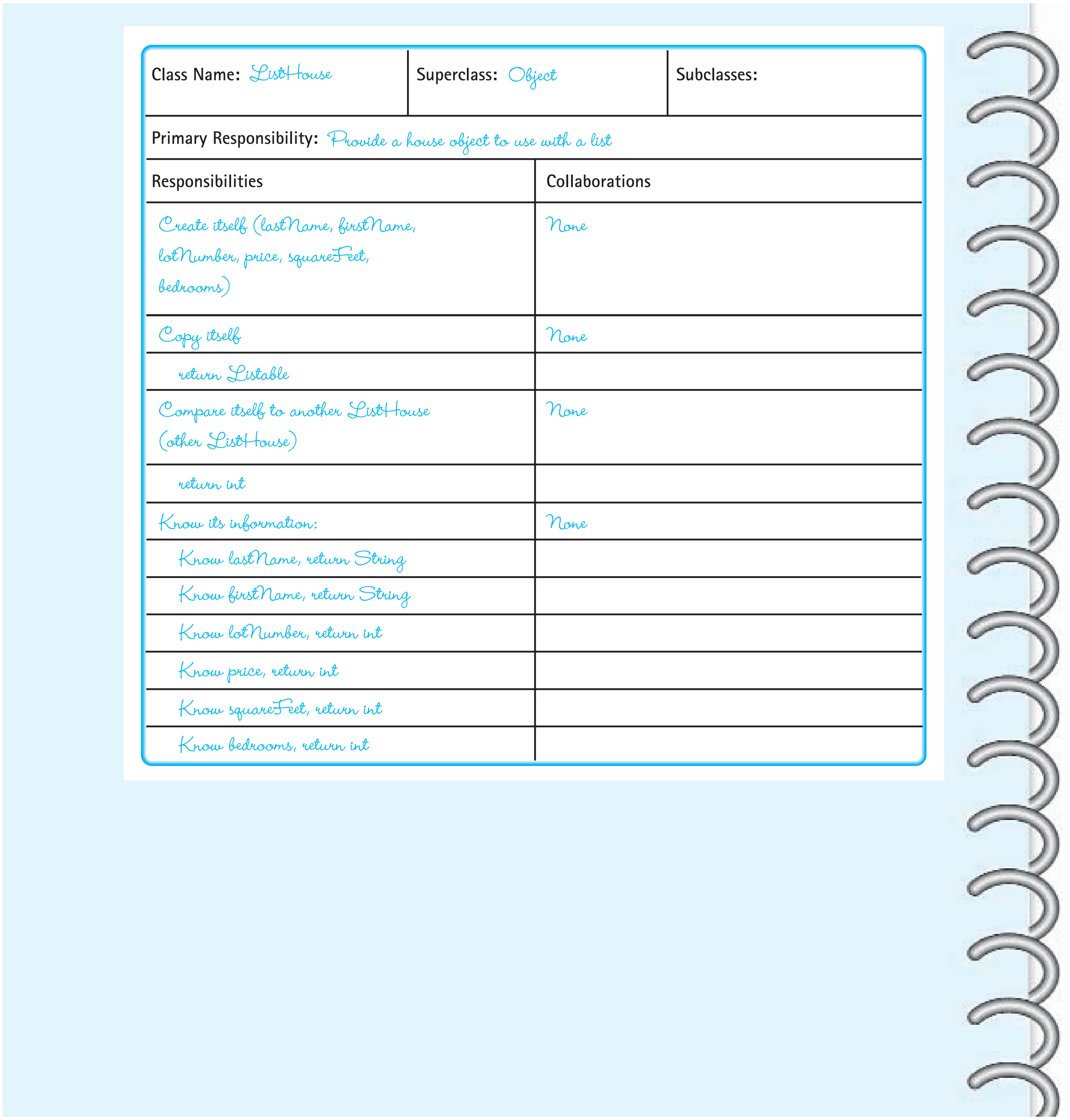
Now we must decide how the `RealEstate` class gets the house information from the `HouseFile` class. Should the information be sent one field at a time or one house at a time? We decide to use the latter approach, since we have already defined a class that encapsulates house data, namely the `ListHouse` class. The `HouseFile` class can provide information to the `RealEstate` class in the form of `ListHouse` objects. And vice versa at the end of the program's execution, the `HouseFile` class can receive information from the `RealEstate` class in the form of `ListHouse` objects.

As our scenario continues we imagine the RealEstate class requesting house information from the HouseFile class. First it informs the HouseFile class that it wishes to begin reading house data. A standard name for this method is reset. Next, as long as there is more house data available, it asks the HouseFile class for data about another house. Therefore, HouseFile must provide both a moreHouses method that returns a boolean, and a getNextHouse method that returns an object of type ListHouse. A similar analysis of how the data can be saved to the file at the end of the program run leads to the identification of a rewrite method and a putToFile method. We also need a method to inform the HouseFile class that we are finished with the file and it should be closed. Finally, we decide to use our standard approach for reading from a file, so we note that HouseFile must collaborate with Java's BufferedReader, FileReader, PrintWriter, and FileWriter classes. We update the CRC card for HouseFile, and move ahead.

As the scenario unfolds we find that most of the operations needed for normal processing have already been assigned to one of our classes. The user clicks on the Clear button and the RealEstate class clears the information from the text fields; the user enters the lot number 45678 into the Lot Number text field and clicks on the Find button; the RealEstate class creates a ListHouse object with 45678 as its lot number, uses the list isThere operation to see if the house is on the list; if it is on the list then the list retrieve operation is used to obtain all of the house information, which is subsequently displayed in the text fields.

But what if the house is not found on the list? When doing scenario analysis it is important to consider all the variations of the scenario. In this case, the program should report to the user that the house was not found. How does it do that? We have uncovered a hole in our interface design. We need to include a way to communicate the results of operations to the user. So, we go back and rework our draft of the interface to include a status box in the upper left corner of the window. We decide we can use this status box to display a message in response to each option selected by the user. For example, if the user selects the Add button and the house is successfully added to the list, we display the message "House added to list." Now our interface is a 10×2 grid.

The investigation of other scenarios is left to the reader. The final set of CRC cards, created strictly for this application, is shown below. We do not include a card for the List ADT since that was not created for this application. Also, we do not include a card for the RealEstate main program, since that is the application that uses the classes represented by the other cards.



Class Name: ListHouse	Superclass: Object	Subclasses:

Primary Responsibility: Provide a house object to use with a list

Responsibilities	Collaborations
Create itself (lastName, firstName, lotNumber, price, squareFeet, bedrooms)	None
Copy itself	None
return Listable	
Compare itself to another ListHouse (other ListHouse)	None
return int	
Know its information:	None
Know lastName, return String	
Know firstName, return String	
Know lotNumber, return int	
Know price, return int	
Know squareFeet, return int	
Know bedrooms, return int	

Class Name: HouseFile	Superclass: Object	Subclasses:
Primary Responsibility: Manage the file of house information		

Responsibilities	Collaborations
Set up for reading	BufferedReader, FileReader
know if there are more houses to read	None
return boolean	
Get the info about the next house from the file	BufferedReader, Integer
return ListHouse	
Set up for writing	PrintWriter, FileWriter
Put info about a house to the file (house)	PrintWriter
Close the file	BufferedReader, PrintWriter

We now turn our attention to the design, implementation, and testing of the identified classes. Note that the `SortedList` class that we use has already been created and tested, so we can assume that it works properly. This is a prime benefit of creating ADTs—once created and tested they can be used with confidence in other systems. We create a package, `ch03.houses`, to hold the `ListHouse` and `HouseFile` classes. We use the package to hold the "helper" classes only; therefore, we do not include the `RealEstate` class, which is an application, as part of the package. You can find the `RealEstate.java` file in the `ch03` subdirectory of the `bookFiles` directory and the `ListHouse.java` and `HouseFile.java` files in the `houses` subdirectory of the `ch03` subdirectory. The files are available on our web site.

The ListHouse Class `ListHouse` must encapsulate house information and it must implement the `Listable` interface, since `ListHouse` objects are placed on a list. Its implementation is rather straightforward. It follows the same patterns established in the `ListCircle` and `ListString` classes developed earlier in the chapter. We must implement `compareTo` and `copy`, but we must also declare variables for all of the information about the house. That is, we must have instance variables for the last name, the first name, the lot number, the price, the number of square feet, and the number of bedrooms. We also need to have observer operations for each of these variables.

```
//----------------------------------------------------------------------------
// ListHouse.java            by Dale/Joyce/Weems                    Chapter 3
//
// Provides elements for a list of house information
//----------------------------------------------------------------------------

package ch03.houses;

import ch03.genericLists.*;

public class ListHouse implements Listable
{
  // House information
  private String lastName;
  private String firstName;
  private int lotNumber;
  private int price;
  private int squareFeet;
  private int bedRooms;

  public ListHouse(String lastName, String firstName, int lotNumber,
                   int price, int squareFeet, int bedRooms )
  {
    this.lastName   = lastName;
    this.firstName  = firstName;
    this.lotNumber  = lotNumber;
    this.price      = price;
    this.squareFeet = squareFeet;
    this.bedRooms   = bedRooms;
  }

  public Listable copy()
  // Returns a copy of this ListHouse
  {
    ListHouse result = new ListHouse(lastName, firstName, lotNumber, price,
                                     squareFeet, bedRooms);
    return result;
  }

  public int compareTo(Listable otherListHouse)
  // Houses are compared  based on their lot numbers
```

```
    {
     ListHouse other = (ListHouse)otherListHouse;
     return (this.lotNumber - other.lotNumber);
    }

    // Observers
    public String lastName()
    {
      return lastName;
    }

    public String firstName()
    {
      return firstName;
    }

    public int lotNumber()
    {
      return lotNumber;
    }

    public int price()
    {
      return price;
    }

    public int squareFeet()
    {
      return squareFeet;
    }

    public int bedRooms()
    {
      return bedRooms;
    }
  }
```

We should test the `ListHouse` class by itself and integrated with the `SortedList` class. We can test it by itself by creating a `TDListHouse` program, similar to the other test driver programs we have used. Our test cases first invoke the constructor, followed by calls to each of the observer methods to ensure that they return the correct information. This could be followed by a test of the copy operation, using it to create a copy of the original `ListHouse` and then repeating the observer method tests on the new object. Finally, the `compareTo` operation must be tested in a variety of situations: compare houses with lot numbers that are less than, equal to, or greater than each other, compare a house to itself, compare a house to a copy of itself, and so on.

`ListHouse` can be tested with the `SortedList` class by repeating the sequence of tests previously used to test our sorted list of strings, replacing the strings with house information.

The HouseFile Class

This class manages the `houses.dat` file. When requested, it pulls data from the file, encapsulates the data into `ListHouse` objects, and returns the `ListHouse` objects to its client. Additionally, it takes `ListHouse` objects from its client and saves the information to the `houses.dat` file.

There is no need to create numerous `HouseFile` objects. Since the class always deals with the same file, we would not want to have several instances of the class interacting with the file at the same time. If we allowed that the file could become corrupted and the system could crash (for example if one instance of the class was trying to read from the file while another instance of the class was trying to write to the file). Therefore, we do not support objects of the class `HouseFile`. We code all of its methods as `static` methods. Recall that this means that the methods are invoked directly through the class itself, as opposed to being invoked through an object of the class. We also declare all of its variables to be `static` variables, that is, class variables as opposed to object variables.

A study of the CRC card for `HouseFile` combined with the analysis of the previous paragraph leads to the following abstract specification of the `HouseFile` class:

House File Specification

Structure:

The house information is kept in a text file called `houses.dat`. For each house the following information is kept, in the order listed, one piece of information per line: last name (`String`), first name (`String`), lot number (`int`), price (`int`), square feet (`int`), and number of bedrooms (`int`).

Operations:

static void reset

Effect:	Resets the file for reading
Throws:	`IOException`

static void rewrite

Effect:	Resets the file for writing
Throws:	`IOException`

static boolean moreHouses

Effect:	Determines whether there is still more house information to be read
Postcondition:	Return value = (there is more house information)

static ListHouse getNextHouse

Effect:	Reads the next house information from the file
Postcondition:	Return value = (a `ListHouse` object containing the next house information)
Throws:	`IOException`

static void putToFile (ListHouse house)

Effect:	Writes the house information to the file
Throws:	`IOException`

static void close

Effect:	Closes the file
Throws:	`IOException`

Reading information from a file and writing information to a file was used in the `TDIncDate` program at the end of Chapter 1. The Java Input/Output feature section that accompanied that program addresses the Java code used to provide those operations. That program only needs to read and write information of type `int`. The `HouseFile` class also performs input and output of `String` information. This is straightforward, since the methods provided by the `java.io` class directly support strings:

```
firstName = inFile.readLine();        // Input of String
outFile.printLn(house.firstName());   // Output of String
```

The `HouseFile` class must keep track of whether or not the `houses.dat` file is closed or opened, and if open, whether it is open for reading or open for writing. It must not allow reading from the file when it is open for writing; nor writing to the file when it is open for reading; nor reading or writing if the file is closed. The `boolean` class variables `inFileOpen` and `outFileOpen` are used to keep track of the status of the file.

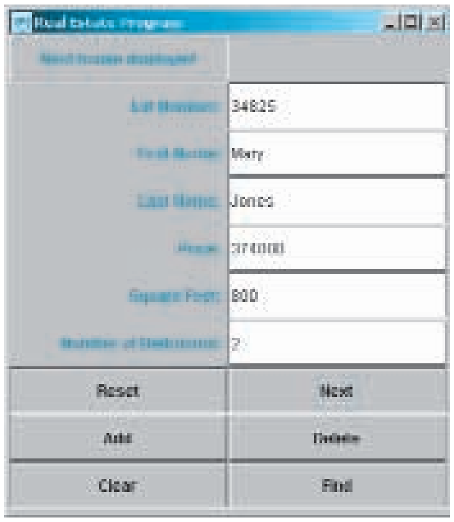
Here is the implementation:

```
//----------------------------------------------------------------------------
// HouseFile.java             by Dale/Joyce/Weems                   Chapter 3
//
// Manages file "houses.dat" of real estate information
//----------------------------------------------------------------------------

package ch03.houses;

import java.io.*;

public class HouseFile
// Manages file "houses.dat" of real estate information
```

```
{
  private static BufferedReader inFile;
  private static PrintWriter outFile;
  private static boolean inFileOpen = false;
  private static boolean outFileOpen = false;
  private static String inString ="";          // Holds "next" line from file
                                               // Equals null if at end of file

  public static void reset() throws IOException
  // Reset file for reading
  {
    if (inFileOpen) inFile.close();
    if (outFileOpen) outFile.close();
    inFile = new BufferedReader(new FileReader("houses.dat"));
    inFileOpen = true;
    inString = inFile.readLine();
  }

  public static void rewrite() throws IOException
  // Reset file for writing
  {
    if (inFileOpen) inFile.close();
    if (outFileOpen) outFile.close();
    outFile = new PrintWriter(new FileWriter("houses.dat"));
    outFileOpen = true;
  }

  public static boolean moreHouses()
  // Returns true if file open for reading and there is still more house
  // information available in it
  {
    if (!inFileOpen || (inString == null))
      return false;
    else return true;
  }

  public static ListHouse getNextHouse() throws IOException
  // Gets and returns house information from the house info file
  // Precondition: inFile is open and holds more house information
  {
    String lastName = "xxxxx";
    String firstName = "xxxxx";
    int lotNumber = 0;
    int price = 0;
    int squareFeet = 0;
    int bedRooms =0;
```

```
      lastName = inString;
      firstName = inFile.readLine();
      lotNumber = Integer.parseInt(inFile.readLine());
      price = Integer.parseInt(inFile.readLine());
      squareFeet = Integer.parseInt(inFile.readLine());
      bedRooms = Integer.parseInt(inFile.readLine());

      inString = inFile.readLine();

      ListHouse house = new ListHouse(lastName, firstName, lotNumber, price,
                                      squareFeet, bedRooms);
      return house;
    }

    public static void putToFile(ListHouse house) throws IOException
    // Puts parameter house information into the house info file
    // Precondition: outFile is open
    {
      outFile.println(house.lastName());
      outFile.println(house.firstName());
      outFile.println(house.lotNumber());
      outFile.println(house.price());
      outFile.println(house.squareFeet());
      outFile.println(house.bedRooms());
    }

    public static void close() throws IOException
    // Closes house info file
    {
      if (inFileOpen) inFile.close();
      if (outFileOpen) outFile.close();
      inFileOpen = false;
      outFileOpen = false;
    }
  }
```

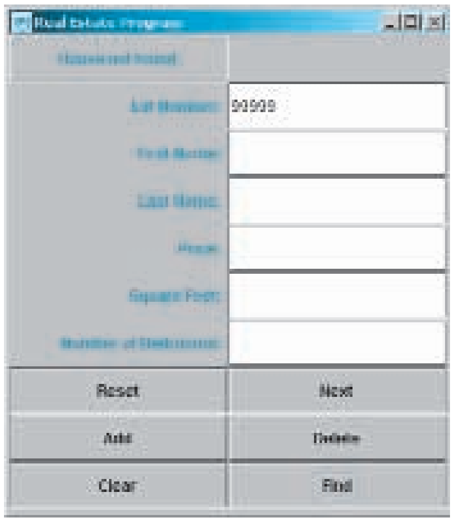
RealEstate Program We now look at the main program, the program that uses all of the other classes to solve the problem. The main program includes the user interface code, in fact that code makes up the majority of the program. Any input/output mechanisms used here that have not yet been encountered in this text are addressed in the feature section, Java Input/Output II that follows the case study. Here is a screen shot of the running program after the user has selected the option to display the "next" house:



Next house displayed

Lot Number: 34825
First Name: Mary
Last Name: Jones
Price: 374000
Square Feet: 800
Number of Bedrooms: 2

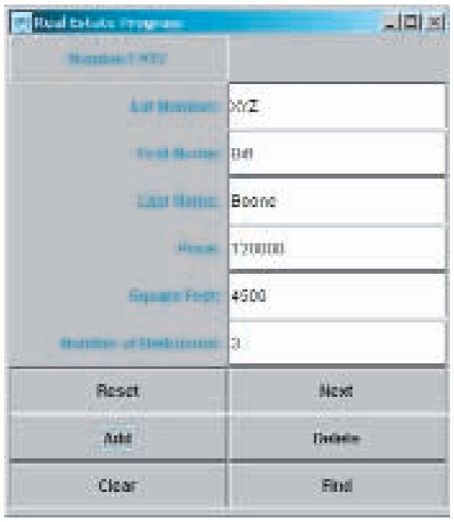
Here is how the program reacts to an attempt to “find” a house that is not on the list:



House not found

Lot Number: 99909
First Name:
Last Name:
Price:
Square Feet:
Number of Bedrooms:

This example shows what happens if the user tries to “add” a house with poorly formatted information:


Real Estate Program

[illegible]

Lot Number: XYZ
First Name: Bill
Last Name: Boone
Price: 170000
Square Feet: 4500
Number of Bedrooms: 3

The algorithm for the main program is as follows:

```
Get the house information from the HouseFile object and build the list of houses.
Present the initial frame
As long as the frame remains open
  Listen for and respond to user choices
    Reset – reset the list and display the first house from the list
    Next – display the next house from the list
    Add – if it is not already on the list, add the currently displayed house to the list
    Delete – if it is on the list, remove the house that matches the currently displayed
             lot number from the list
    Clear – clear the text fields
    Find – display the house from the list that matches the currently displayed lot
           number, if possible
Send the information about the houses from the list to the HouseFile object
```

Processing begins by using the `HouseFile` class to obtain the house information from the file, and using the `SortList` class to store the house information. This is accomplished through the following steps:

```
Create a new list
Reset the house file for reading
while there are still more houses to read
  Read the next house and
  Insert it into the list
```

The code that corresponds to this algorithm can be found in the main method, after the various interface labels, text fields, and buttons have been set up, and the display frame has been initialized.

So, that's how we get the information from the file onto the list at the beginning of our processing. How do we reverse this process. That is, how do we take the information from the list and save it back to the file? Actually, the save algorithm is very similar:

```
Reset the house file for writing
Rest the list
for each house on the list
  Get the house from the list and
  Store it in the file
```

Where should the code for this algorithm go in the program? We want this code to be one of the last things that the program does. Remember, this is an event-driven program, so we cannot just put the code at the end of the `main` method and expect it to be executed last. Instead, we define our own `WindowClosing` event handler and place the corresponding code there. In this way, when the user is finished and closes the application window, the information is saved.

Now that we have determined how we get the house information from the file to the list, and vice versa, the only processing that remains is what occurs in response to the user pressing buttons in the interface. There are six buttons. The processing required by each button is fairly well stated in the original algorithm above. Many of them require moving information from the display (the set of text fields) to the list, or vice versa. This leads us to design three helper methods

- `showHouse` – accepts a `ListHouse` object as a parameter and displays the information about the object in the text fields
- `getHouse` – obtains the information from the textboxes, turns it into a `ListHouse` object, and returns the object
- `clearHouse` – clears the information from the textboxes

The implementation of these methods is straightforward, and with their help we can implement the button-processing routines without much difficulty. For example, the algorithm to handle the Reset button is:

```
Reset the list
if the list is empty
  clearHouse
else
  Set house to the first house on the list
  showHouse(house)
Report "List reset" through the status label
```

The code that corresponds to this algorithm is placed in the `ActionHandler` class, and associated with the "Reset" event.

Study the code for the other event handlers to see how they use the helper methods to perform their tasks. Note that each of the event handlers that depends upon the user entering information into a text field, use Java's exception handling mechanism to protect the application from user input errors. For example, the algorithm for the Add event is:

```
try
  Set house to getHouse
  if house is already on the list
    Report "Lot number already in use" through the status label
  else
    Insert house into the list
    Report "House added to list" through the status label
catch NumberFormat Exception
  Report a problem with the house data through the status label
```

Since four of the house fields require `int` data, if the system raises a `NumberFormatException` it is because something other than an integer was listed in at least one of those fields. Therefore, the message displayed through the status label is "Number?", followed by an echo of the bad data. The bad data value is available through the exception object's `getMessage` method. You can see that similar protection is provided for the Delete and Find event handlers in the code.

Here is the listing for the Real Estate application:

```
//----------------------------------------------------------------------------
// RealEstate.java             by Dale/Joyce/Weems                   Chapter 3
//
// Helps keep track of a company's real estate listings
//----------------------------------------------------------------------------

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.io.*;
import ch03.houses.*;
import ch03.genericLists.*;

public class RealEstate
{
  // The list of house information
  private static SortedList list = new SortedList();

  // Text fields
  private static JTextField lotText;              // Lot number field
  private static JTextField firstText;            // First name field
  private static JTextField lastText;             // Last name field
  private static JTextField priceText;            // Price field
  private static JTextField feetText;             // Square feet field
  private static JTextField bedText;              // Number of bedrooms field

  // Status Label
  private static JLabel statusLabel;              // Label for status info

  // Display information about parameter house on screen
  private static void showHouse(ListHouse house)
  {
    lotText.setText(Integer.toString(house.lotNumber()));
    firstText.setText(house.firstName());
    lastText.setText(house.lastName());
    priceText.setText(Integer.toString(house.price()));
    feetText.setText(Integer.toString(house.squareFeet()));
    bedText.setText(Integer.toString(house.bedRooms()));
  }

  // Returns current screen information as a ListHouse
  private static ListHouse getHouse()
  {
    String lastName;
    String firstName;
    int lotNumber;
    int price;
    int squareFeet;
    int bedRooms;

    lotNumber = Integer.parseInt(lotText.getText());
    firstName = firstText.getText();
    lastName = lastText.getText();
    price = Integer.parseInt(priceText.getText());
    squareFeet = Integer.parseInt(feetText.getText());
    bedRooms = Integer.parseInt(bedText.getText());

    ListHouse house = new ListHouse(lastName, firstName, lotNumber, price,
                                    squareFeet, bedRooms);
    return house;
  }

  // Clears house information from screen
  private static void clearHouse()
  {
    lotText.setText("");
    firstText.setText("");
```

```
      lastText.setText("");
      priceText.setText("");
      feetText.setText("");
      bedText.setText("");
    }

    // Define a button listener
    private static class ActionHandler implements ActionListener
    {
      public void actionPerformed(ActionEvent event)
      // Listener for the button events
      {
        ListHouse house;

        if (event.getActionCommand().equals("Reset"))
        { // Handles Reset event
          list.reset();
          if (list.lengthIs() == 0)
            clearHouse();
          else
          {
            house = (ListHouse)list.getNextItem();
            showHouse(house);
          }
          statusLabel.setText("List reset");
        }
        else
        if (event.getActionCommand().equals("Next"))
        { // Handles Next event
          if (list.lengthIs() == 0)
            statusLabel.setText("list is empty!");
          else
            {
            house = (ListHouse)list.getNextItem();
            showHouse(house);
            statusLabel.setText("Next house displayed");
          }
        }
        else
        if (event.getActionCommand().equals("Add"))
        { // Handles Add event
          try
          {
            house = getHouse();
            if (list.isThere(house))
              statusLabel.setText("Lot number already in use");
            else
            {
              list.insert(house);
              statusLabel.setText("House added to list");
            }
          }
          catch (NumberFormatException badHouseData)
          {
            // Text field info incorrectly formated
            statusLabel.setText("Number? " + badHouseData.getMessage());
          }
        }
        else
        if (event.getActionCommand().equals("Delete"))
        { // Handles Delete event
          try
          {
            house = getHouse();
            if (list.isThere(house))
            {
              list.delete(house);
              statusLabel.setText("House deleted");
            }
            else
              statusLabel.setText("Lot number not on list");
          }
          catch (NumberFormatException badHouseData)
          {
            // Text field info incorrectly formated
            statusLabel.setText("Number? " + badHouseData.getMessage());
          }
        }
        else
        if (event.getActionCommand().equals("Clear"))
        { // Handles Clear event
          clearHouse();
          statusLabel.setText(list.lengthIs() + " houses on list");
        }
        else
        if (event.getActionCommand().equals("Find"))
        { // Handles Find event
          int lotNumber;
          try
```

```
            {
              lotNumber = Integer.parseInt(lotText.getText());
              house = new ListHouse("", "", lotNumber, 0, 0, 0);
              if (list.isThere(house))
              {
                house = (ListHouse)list.retrieve(house);
                showHouse(house);
                statusLabel.setText("House found");
              }
              else
              statusLabel.setText("House not found");
            }
            catch (NumberFormatException badHouseData)
            {
              // Text field info incorrectly formated
              statusLabel.setText("Number? " + badHouseData.getMessage());
            }
          }
        }
      }

      public static void main(String args[]) throws IOException

      {
        ListHouse house;
        char command;
        int length;

        JLabel blankLabel;          // To use up one frame slot

        JLabel lotLabel;            // Labels for input fields
        JLabel firstLabel;
        JLabel lastLabel;
        JLabel priceLabel;
        JLabel feetLabel;
        JLabel bedLabel;

        JButton reset;              // Reset button
        JButton next;               // Next button
        JButton add;                // Add button
        JButton delete;             // Delete button
        JButton clear;              // Clear button
        JButton find;               // Find button
        ActionHandler action;       // Declare listener
```

```
// Declare/Instantiate/Initialize display frame
JFrame displayFrame = new JFrame();
displayFrame.setTitle("Real Estate Program");
displayFrame.setSize(350,400);
displayFrame.addWindowListener(new WindowAdapter()  // handle window
                                                    //   closing
{
  public void windowClosing(WindowEvent event)
  {
    ListHouse house;
    displayFrame.dispose();                     // Close window
    try
    {
      // Store info from list into house file
      HouseFile.rewrite();
      list.reset();
      int length = list.lengthIs();
      for (int counter = 1; counter <= length; counter++)
      {
        house = (ListHouse)list.getNextItem();
        HouseFile.putToFile(house);
      }
      HouseFile.close();
    }
    catch (IOException fileCloseProblem)
    {
      System.out.println("Exception raised concerning the house info file "
                          + "upon program termination");
    }
    System.exit(0);                           // Quit the program
  }
});

// Instantiate content pane and information panel
Container contentPane = displayFrame.getContentPane();
JPanel infoPanel = new JPanel();

// Instantiate/initialize labels, and text fields
statusLabel = new JLabel("", JLabel.CENTER);
statusLabel.setBorder(new LineBorder(Color.red));
blankLabel = new JLabel("");
lotLabel = new JLabel("Lot Number:  ", JLabel.RIGHT);
lotText = new JTextField("", 15);
firstLabel = new JLabel("First Name:  ", JLabel.RIGHT);
firstText = new JTextField("", 15);
```

```
lastLabel = new JLabel("Last Name:  ", JLabel.RIGHT);
lastText = new JTextField("", 15);
priceLabel = new JLabel("Price:  ", JLabel.RIGHT);
priceText = new JTextField("", 15);
feetLabel = new JLabel("Square Feet:  ", JLabel.RIGHT);
feetText = new JTextField("", 15);
bedLabel = new JLabel("Number of Bedrooms:  ", JLabel.RIGHT);
bedText = new JTextField("", 15);

// Instantiate/register buttons
reset = new JButton("Reset");
next = new JButton("Next");
add = new JButton("Add");
delete = new JButton("Delete");
clear = new JButton("Clear");
find = new JButton("Find");

// Instantiate/register button listeners
action = new ActionHandler();
reset.addActionListener(action);
next.addActionListener(action);
add.addActionListener(action);
delete.addActionListener(action);
clear.addActionListener(action);
find.addActionListener(action);

// Load info from house file into list
HouseFile.reset();
while (HouseFile.moreHouses())
{
  house = HouseFile.getNextHouse();
  list.insert(house);
}

// If possible insert info about first house into text fields
list.reset();
if (list.lengthIs() != 0)
{
  house = (ListHouse)list.getNextItem();
  showHouse(house);
}

// Update status
statusLabel.setText(list.lengthIs() + " houses on list                ");
```

```
      // Add components to frame
      infoPanel.setLayout(new GridLayout(10,2));
      infoPanel.add(statusLabel);
      infoPanel.add(blankLabel);
      infoPanel.add(lotLabel);
      infoPanel.add(lotText);
      infoPanel.add(firstLabel);
      infoPanel.add(firstText);
      infoPanel.add(lastLabel);
      infoPanel.add(lastText);
      infoPanel.add(priceLabel);
      infoPanel.add(priceText);
      infoPanel.add(feetLabel);
      infoPanel.add(feetText);
      infoPanel.add(bedLabel);
      infoPanel.add(bedText);
      infoPanel.add(reset);
      infoPanel.add(next);
      infoPanel.add(add);
      infoPanel.add(delete);
      infoPanel.add(clear);
      infoPanel.add(find);

      // Set up and show the frame
      contentPane.add(infoPanel);
      displayFrame.show();
    }
  }
```

Test Plan We assume classes `Listable` and `SortedList` have been thoroughly tested. This leaves classes `ListHouse`, `HouseFile`, and the Real Estate application program to test. To test the two classes we could create test driver programs to call the various methods and display results. But recall that these classes were created specifically for the Real Estate application. Therefore, we can use the main application as the test driver to test them. In other words, we can test everything together.

The first task is to create a master file of houses by using the Add command to input several houses and quit. We then need to input a variety of commands to add more houses, delete houses, find houses, and look through the list of houses with the Reset and Next buttons. We should try the operations with good data and with bad data (for example nonintegral lot numbers) We should try the operations in as many different sequences as we can devise. The program must be run several times in order to test the access and preservation of the data base (file `houses.dat`). We leave the final test plan as an exercise.

In the discussion of object-oriented design in Chapter 1, we said that the code responsible for coordinating the objects is called a driver. Now, we can see why. A driver program in testing terminology is a program whose role is to call various subprograms and observe their behavior. In object-oriented terminology, a program is a collection of collaborating objects. Therefore, the role of the main application is to invoke operations on certain objects, that is, get them started collaborating, so the term driver is appropriate. In subsequent chapters, when we use the term driver, the meaning should be clear from the context.

Java Input/Output II

Let's look at the graphical user interface of the Real Estate program. This interface is more complicated than that used by the test driver program we saw in Chapter 1. The test driver displayed only labels, whereas the Real Estate program displays labels, text fields, and buttons. However, the biggest difference is in how the frame is used by the user of the program. The test driver program simply displayed a few lines of information on its frame, and then waited for the user to close the frame. The frame for the Real Estate program, on the other hand, is changed based on actions performed by the user. It is truly interactive.

Throughout the following discussion, please review the code from the Real Estate program that corresponds to the particular discussion topic.

The Frame

First let's look at how the frame is constructed. The setup of the frame is similar to that performed by the test driver program. However, handling window closing is more complicated here, since we must perform some special processing (save the information from the list to the `houses.dat` file) instead of just exiting the system. We name our frame `displayFrame` and set its title and size as we did before.

```
JFrame displayFrame = new JFrame();
displayFrame.setTitle("Real Estate Program");
displayFrame.setSize(350,400);
```

Next we define the needed reaction to the window-closing event with the following commands (see the main method, after a sequence of label and button declarations):

```
displayFrame.addWindowListener(new WindowAdapter()   // Handle window
                                                     //   closing
{
. . .
  });
```

The actual code executed when the window is closed is represented by the "..."; it saves the current list of house information to the data file, and then exits the program. Let's discuss the `addWindowListener` method. As you know, when the frame is displayed, it appears in its own window. Normally, when you define a window listener from within a Java program, you must define how the window reacts to various events: closing the window, resizing the window, activating the window, and so on. You must define methods to handle all of these events. However, in our program we only want to handle one of these events, the window closing event. The code above lets us directly handle the window closing event while we accept default "do nothing" handlers for all the other window events. In effect, a `WindowAdapter` object is a window that has "do nothing" events defined for all window events. We are adding a "window closing listener" to our frame that tells the program what processing to perform when someone closes the window, overriding the default "do nothing" event handler in this case.

As was done for the test driver, we next instantiate the content pane, and an information panel:

```
Container contentPane = displayFrame.getContentPane();
JPanel infoPanel = new JPanel();
```

Recall that the content pane is the part of a frame that is used to display information generated by a program, and a panel is a container, capable of holding other constructs, where the program organizes its information for display.

Components

Next we create components that are eventually added to our panel. We create labels, text fields, and buttons. We look at each in turn, starting with labels. You are familiar with labels from the Chapter 1 test driver. In the Real Estate program, we exploit a little more of the functionality provided by the `JLabel` class. Consider the two statements that set up the status label—the label used in the interface to display a message describing the result of a user action:

```
statusLabel = new JLabel("", JLabel.CENTER);
statusLabel.setBorder(new LineBorder(Color.red));
```

In addition to passing the `JLabel` constructor an initial string, we pass it the constant `CENTER`, defined in the `JLabel` class. This sets the label so that it displays text centered in the area allocated to it. That property persists until we change it with a call to the label's `setHorizontalAlignment` method. We follow the instantiation of `statusLabel` with a message to it, to set its border to a line border with the color red. Borders can be set for any Java Swing component that extends the `JComponent` class; that is, for most Swing components. Swing supports eight kinds of borders—we have elected to use the line border in this case. Note that we pass the `LineBorder` constructor a constant of the `Color` class. Also note, that to use borders, we must import `javax.swing.border.*` into the program. Finally, note that most of the labels used by the program are declared at the beginning of the main method, but the `statusLabel` label is declared outside the main method, since it needs to be visible to some of the helper methods.

Intermingled with the label instantiations are instantiations of text fields. This is a new construct for us. A text field is a box that allows the user to enter a single line of text. In the Real Estate program, we use them to both gather and present information to the user. An example of a test field instantiation is:

```
lotText = new JTextField("", 15);
```

The string parameter sets the initial text in the text field (in this case to the empty string); the integer parameter sets the width of the text field. Since all of our text fields are accessed by helper methods, they are all declared outside of the main method.

The text displayed in a text field can be changed with the `setText` method, as is done in the `showHouse` helper method to display the information about a house on the interface. Of course, the user can directly enter text into a text field box and change its contents. The `getText` method is used by a program to obtain the current information in a text field. See the `getHouse` helper method for examples of its use.

The final construct used in our interface is the button. Buttons are used to generate events, when pressed by the user. Button definition is easy. Just invoke the button constructor, passing it the string to be displayed on the face of the button, as follows:

```
reset = new JButton("Reset");
```

Although button-related events are handled by helper methods, the buttons themselves are only used within the `main` method, so all button declarations are at the beginning of `main`. There are six buttons altogether. The Real Estate program "listens" for its user to press one of the buttons, performs processing related to the pressed button, updates the frame appropriately, and then resumes listening. To understand how this works, and how we implement it, we must look at the Java event model.

The Event Model

In an event-driven Java program, there are two important entities, event sources and event listeners. The sources generate an event, usually due to some action by the user. The listeners are waiting for certain events to occur, and when they do, they react to the event by performing some related processing. In our program, the `JButton` object `reset` is an event source, and the `ActionHandler` object `action` is an event listener. These objects are declared and instantiated by the statements:

```
JButton reset;
reset = new JButton("Reset");

ActionHandler action;
action = new ActionHandler();
```

We have already examined the `JButton` statements. But, what is an `ActionHandler`? You won't find it defined in any Java library documentation because it is a class created just for the Real Estate program. It is an inner class. You can find its definition in the program listing just after the helper methods that manage the house information displayed on the screen. It looks like this (with many lines deleted):

```
private static class ActionHandler implements ActionListener
  {
    public void actionPerformed(ActionEvent event)
```

```
// Listener for the button events
{
  . . .
  if (event.getActionCommand().equals("Reset"))
  { // Handles Reset event
    . . .
  }
  else
  if (event.getActionCommand().equals("Next"))
  { // Handles Next event
    . . .
  }
  . . .
}
```

As you can see, `ActionHandler` implements the `ActionListener` interface. Therefore, `action` is also an `ActionListener`. Action listeners are just one of several Java listener types. Another example is window listeners, which we use to close our frames. We use action listeners for our user interfaces. We return to the definition of `ActionHandler` below. First, let's see how we "connect" the event source and the event listener.

The `action` listener is registered with the `reset` button with the command

```
reset.addActionListener(action);
```

As you can see in the program code, the action listener is also registered with the other five buttons.

The registration of an event listener with an event source means that whenever an event occurs to the event source, such as a button being pressed, an announcement of the event is passed to the event listener. There are all sorts of events supported by Java. In our case we are only interested in "action" events, a subset of the set of all potential events, so we use an `ActionListener` listener.

How does the event source send "an announcement" of an event to the listener? Through a call to one of the listener's methods, of course. In the case of action events, the source calls the listener's `actionPerformed` method, and passes it an `ActionEvent` object that represents the event that occurred. The `ActionListener` interface declares an abstract `actionPerformed` method, so we know that any class that implements `ActionListener`, like our `ActionHandler` class, must provide an implementation of `actionPerformed`. You do not see a call to the `actionPerformed` method anywhere in our program. We do not explicitly invoke the method in our code; it is automatically called when a button is pressed by the user. Such a method invocation is sometimes called an *implicit invocation*.

Let's review. In the Real Estate program, we have six buttons that are event sources. We have one event listener, *action*, which has been registered through `addActionListener` to all six buttons. When any of the buttons are pressed the `actionPerformed` method of the `action` object is invoked, and passed an `event` object that represents the button-pressed event. At that point, the `actionPerformed` method must respond to the event. How does it do that?

Look again at the code listed above for the `ActionHandler` class. You can see that the `actionPerformed` method is implemented as a series of *if-else* statements.

```
if (event.getActionCommand().equals("Reset"))
{ // Handles Reset event
  . . .
}
else
```

Each *if*-block handles a different button being pressed. The boolean expressions use the `getActionCommand` method of the `ActionEvent` class to obtain a string signifying the specific real event that the `event` object represents. In the case of button pressing events, this string is simply the string displayed on the face of the button. Therefore, when a button is pressed, the appropriate *if*-block is executed. Take a minute to browse the code in the *if*-blocks to see how the program handles each of the buttons being pressed.

Presenting the Interface

Now that we have created the frame, the labels, the text fields, the buttons and associated actions with each of the buttons, we are ready to build and display the interface. We use the same approach we did for the Chapter 1 test driver program. First, we set up a 10 × 2 grid in our panel:

```
infoPanel.setLayout(new GridLayout(10,2));
```

Next we add all of our components to our panel, in the order we want them to appear (left to right, top to bottom):

```
infoPanel.add(statusLabel);
infoPanel.add(blankLabel);
infoPanel.add(lotLabel);
infoPanel.add(lotText);
. . .
infoPanel.add(find);
```

Finally, we add the panel to the frame's content pane, and show the frame:

```
contentPane.add(infoPanel);
displayFrame.show();
```

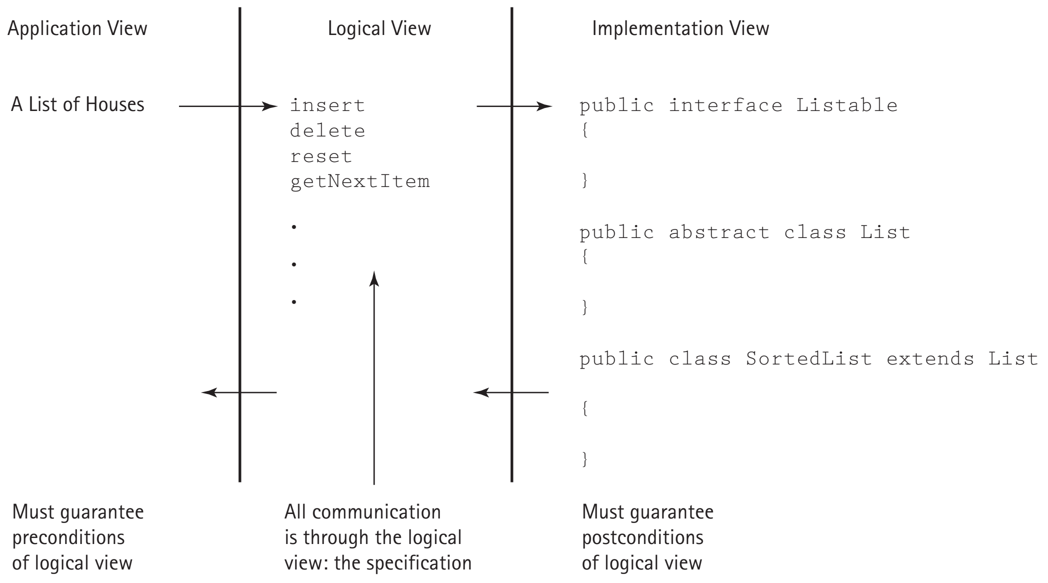
Summary

In this chapter, we have created two abstract data types that represent lists. The Unsorted List ADT assumes that the list elements are not sorted by key; the Sorted List ADT assumes that the list elements are sorted by key. We have viewed each from three perspectives: the logical level, the application level, and the implementation level. The extended Case Study uses the Sorted List ADT to help solve a problem. Figure 3.18 shows the relationships among the three views of the list data in the Case Study.

As we progressed through the chapter we expanded our use of Java constructs to support the list abstractions. In the first part of the chapter, we worked through the following variations of lists:

- `UnsortedStringList`—an unsorted list of strings
- `StringList`—an abstract string list specification; valid for both sorted and unsorted lists
- `UnsortedStringList2`—an extension of `StringList`
- `SortedStringList`—another extension of `StringList`

In order to make the software as reusable as possible, we learned how to use the Java *interface* mechanism to create generic ADTs. The user of the ADT must prepare a class that defines the objects to be in each container class. In the case of the list abstraction, objects to be contained on a list must implement the `Listable` interface; therefore, they must have an appropriate `compareTo` and a `copy` method associated with them. By requiring the user to meet this standard for the objects on the list, the code of the ADTs is very general.



Application View
Logical View
Implementation View
A List of Houses
insert
delete
reset
getNextItem
public interface Listable
{
}
public abstract class List
{
}
public class SortedList extends List
{
}
Must guarantee preconditions of logical view
All communication is through the logical view: the specification
Must guarantee postconditions of logical view

Figure 3.18 *Relationships among the views of data*

The Unsorted List or Sorted List ADT can process items of any kind; they are completely context independent. Within the chapter we saw examples of how to create lists of circles, strings, and houses. The ability to create generic structures led to two more list variations:

- `List`—an abstract list specification, no longer tied to strings; includes a `retrieve` operation
- `SortedList`—an extension of `List`

We compared the operations on the two ADTs using Big-O notation. Insertion into an unsorted list is O(1); insertion into a sorted list is O(N). Deletions from both are O(N). Searching in the unsorted list is O(N); searching in a sorted list is order O(log2N) if a binary search is used.

We have also seen how to write test plans for ADTs.

Summary of Classes and Support Files

The classes and files are listed in the order in which they appear in the text. Inner classes are not included. The package a class belongs to, if any, is listed in parenthesis under Notes. The class and support files are available on our web site. They can be found in the `ch03` subdirectory of the `bookFiles` directory.

Classes, Interfaces, and Support Files Defined in Chapter 3

File	1st Ref.	Notes
`UnsortedStringList.java`	page 150	(`ch03.stringLists`) Array-based implementation of an unsorted string list ADT
`TDUnsortedStringList.java`	page 160	Test driver for `UnsortedStringList.java`
`StringList.java`	page 165	(`ch03.stringLists`) Abstract class—defines all the constructs for an array based list of strings that do not depend on whether or not the list is sorted
`UnsortedStringList2.java`	page 166	(`ch03.stringLists`) Extends `StringList` under the assumption that the list is *not* kept sorted
`SortedStringList.java`	page 181	(`ch03.stringLists`) Extends `StringList` under the assumption that the list *is* kept sorted
`Listable.java`	page 194	(`ch03.genericLists`) Interface—objects used with the following list classes must be derived from classes that implement this interface
`ListCircle.java`	page 195	(`ch03.genericLists`) Example of a class that implements `Listable`

(continued on next page)

File	1st Ref.	Notes
`List.java`	page 197	(`ch03.genericLists`) Abstract class—defines all the constructs for an array-based generic list that do not depend on whether or not the list is sorted; the list stores objects derived from a class that implements `Listable`; includes a `retrieve` method, that was not part of the previous lists
`SortedList.java`	page 200	(`ch03.genericLists`) Extends `List` under the assumption that the list *is* kept sorted
`ListString.java`	page 204	(`ch03.genericLists`) Another example of a class that implements `Listable`
`TDSortedList.java`	page 206	Test driver for `SortedList.java`
`ListHouse.java`	page 215	(`ch03.houses`) Implements `Listable`; provides information about a house that can be stored on a list
`HouseFile.java`	page 218	(`ch03.houses`) Manages the `houses.dat` file
`RealEstate.java`	page 224	The real estate application
`testlist1.dat`	page 162	Test data for the `TDUnsortedStringList` program
`testout1.dat`	page 162	Results of using `testlist1.dat` as input to the Unsorted String List test driver
`testlist2.dat`	page 206	Test data for the `TDSortedList` program
`testout2.dat`	page 206	Results of using `testlist2.dat` as input to the Sorted List test driver

The diagrams in Figure 3.19 show the relationships among the classes listed above. Abstract classes are shown in (*Italics*) within parentheses, interfaces are shown within <brackets>, and applications are $\boxed{\text{boxed}}$. Relationships are shown by arrows using UML standard representations (solid arrow with hollow arrowhead represents the inheritance relationship "extends," dotted arrow with hollow arrowhead represents the implements relationship between a class and an interface, and dotted arrow with open arrowhead represents a "uses" relationship—the latter relationships are also labeled "uses.") Finally, the package groupings are indicated by "blue rectangles."

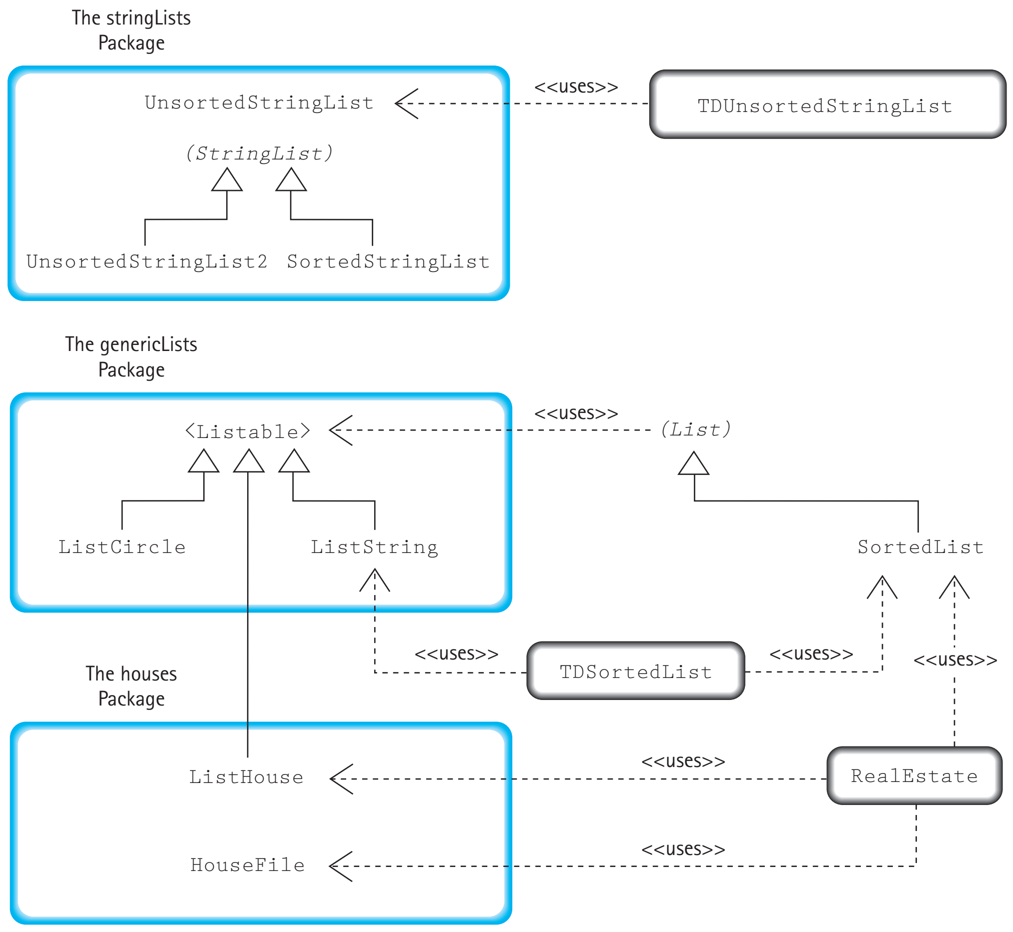


Figure 3.19 *Chapter 3 classes and their relationships*

On page 241 is a list of the Java Library Classes that were used in this chapter for the first time in the textbook. The classes are listed in the order in which they are first used. Note that in some classes the methods listed might not be defined directly in the class; they might be defined in one of its superclasses. With the methods we also list constructors, if appropriate. For more information about the library classes and methods the reader can check Sun's Java documentation.

Library Classes Used in Chapter 3 for the First Time

Class Name	Package	Overview	Methods Used	Where Used
`JTextField`	`swing`	Provides a container for a single line of user text	`getText`, `JTextField`, `setText`	`RealEstate`
`ActionListener`	`awt.event`	An interface for classes that listen for and handle action events		`RealEstate`
`ActionEvent`	`awt.event`	Provides objects for passing event information between event sources and event listeners	`getActionCommand`	`RealEstate`
`JButton`	`swing`	Provides a container for an interface button	`ActionListener`, `JButton`,	`RealEstate`
`LineBorder`	`swing`	Sets a border for the display of a component	`LineBorder`	`RealEstate`
`Color`	`lang`	Provides color constants		`RealEstate`

Exercises

3.1 Lists

1. Give examples from the "real world" of unsorted lists, sorted lists, lists that permit duplicate keys, and lists that do not permit duplicate keys.
2. Describe how the individuals in each of the following groups of people could be uniquely identified; that is, what would make a good key value for each of the groups.
   a. Citizens of a country who are eligible to vote
   b. Members of a sports team
   c. Students in a school
   d. E-mail users
   e. Automobile drivers
   f. Actors/actresses in a play

3.2 Abstract Data Type Unsorted List

3. Classify each of the Unsorted List ADT operations (`UnsortedStringList`, `isFull`, `lengthIs`, `isThere`, `insert`, `delete`, `reset`, `getNextItem`) according to operation type (Constructor, Iterator, Observer, Transformer).
4. The chapter specifies and implements an Unsorted List ADT (for strings).
   a. Design an algorithm for an application-level routine `printLast` that accepts a list as a parameter and returns a `boolean`. If the list is empty, the routine prints "List is empty" and returns `false`. Otherwise, it prints the last item of the list and returns `true`. The signature for the routine should be

```
boolean printLast(PrintWriter outfile, UnsortedStringList list)
```

   b. Devise a test plan for your algorithm.
   c. Implement and test your algorithm.
5. The chapter specifies and implements an Unsorted List ADT (for strings).
   a. Design an algorithm for an application level routine that accepts two lists as parameters, and returns a count of how many items from the first list are also on the second list. The signature for the routine should be

```
int compareLists(UnsortedStringList list1, UnsortedStringList list2)
```

   b. Devise a test plan for your algorithm.
   c. Implement and test your algorithm.
6. What happens if the constructor for `UnsortedStringList` is passed a negative parameter? How could this situation be handled by redesigning the constructor?
7. A friend suggests that since the `delete` operation of the Unsorted List ADT assumes that the parameter element is already on the list, the designers may as well assume the same thing for other operations since it would simplify things. Your friend wants to add the assumption to both the `isThere` and the `insert` operations! What do you think?
8. Describe the ramifications of each of the following changes to the chapter's code for the indicated `UnsortedStringList` methods.

a.	`isFull`	change "return (list.length == numItems);" to "return (list.length = numItems);"
b.	`lengthIs`	change "return numItems;" to "return list.length;"
c.	`isThere`	change the second "moreToSearch = (location < numItems);" to "moreToSearch = (location <= numItems);"
d.	`insert`	remove "numItems++;"
e.	`delete`	remove "numItems–;"

9. The test plan on page 181 for the `UnsortedStringList` class was not complete.
   a. Complete the test plan.
   b. Create a set of test input files that represents the completed test plan.

c. Use the `TDUnsortedStringList` program, available with the rest of the textbook's programs, to run and verify your tests.

10. The Unsorted List ADT (for `UnsortedStringList`) is to be extended with a `boolean` operation, `isEmpty`, which determines whether or not the list is empty.
    a. Write the specifications for this operation.
    b. Write a method to implement the operation.

11. The Unsorted List ADT (for `UnsortedStringList`) is to be extended with an operation, `smallest`, which returns a copy of the "smallest" list element. It is assumed that the operation will not be invoked if the list is empty.
    a. Write the specifications for this operation.
    b. Write a method to implement the operation.

12. Rather than enhancing the Unsorted List ADT by adding a `smallest` operation, you decide to write a client method to do the same task.
    a. Write the specifications for this method.
    b. Write the code for the method, using the operations provided by the Unsorted List ADT
    c. Write a paragraph comparing the client method and the ADT method (Exercise 11) for the same task.

13. The specifications for the Unsorted List ADT `delete` operation state that the item to be deleted is in the list.
    a. Create a specification for a new form of delete, called `tryDelete`, that leaves the list unchanged if the item to be deleted is not in the list. The new delete operation should return a `boolean` value `true` if the item was found and deleted, `false` if the item was not on the list.
    b. Implement `tryDelete` as specified in (a).

14. The specifications for the Unsorted List ADT state that the list contains unique items. Suppose this assumption is dropped, and the list is allowed to contain duplicate items.
    a. How would the specification have to be changed?
    b. Create a specification for a new form of delete for this new ADT, called `deleteAll`, that deletes all list elements that match the parameter item's key. You should still assume that at least one matching item is on the list.
    c. Implement `deleteAll` as specified in (b).

15. The text's implementation of the `delete` operation for the Unsorted List ADT (`UnsortedStringList`) does not maintain the order of insertions because the algorithm swaps the last item into the position of the one being deleted and then decrements length.
    a. Would there be any advantage to having `delete` maintain the insertion order? Justify your answer.

   b. Modify `delete` so that the insertion order is maintained. Code your algorithm, and test it.

16. Change the specifications for the Unsorted List ADT so that `insert` throws an exception if the list is full. Implement the revised specification.

17. Create a new implementation of the Unsorted List ADT (`UnsortedStringList`) using the Java Library's `ArrayList` class instead of plain arrays.

3.3 Abstract Classes

18. The abstract class `StringList` contains both abstract and concrete methods.
    a. List the abstract methods.
    b. List the concrete methods.
    c. Explain the difference between an abstract method and a concrete method.

19. Suppose you wanted to add the operation `isEmpty`, as defined in Exercise 10, to the `StringList` class. Would you make it an abstract method or a concrete method? Justify your answer.

20. Suppose you wanted to add the operation `smallest`, as defined in Exercise 11, to the `StringList` class. Would you make it an abstract method or a concrete method? Justify your answer.

21. Consider the UML diagram in Figure 3.5.
    a. What does the "+" symbol represent?
    b. What does the "#" symbol represent?
    c. What does the arrow represent?
    d. Why are some of the method names italicized?
    e. Why is the variables section of the class diagram for the `Unsorted-StringList2` class empty?

3.4 Abstract Data Type Sorted List

22. The Sorted List ADT (for `SortedStringList`) is to be extended with an operation, `smallest`, which returns a copy of the "smallest" list element. It is assumed that the operation will not be invoked if the list is empty.
    a. Write the specifications for this operation.
    b. Write a method to implement the operation.

23. Rather than enhancing the Sorted List ADT by adding a `smallest` operation, you decide to write a client method to do the same task.
    a. Write the specifications for this method.
    b. Write the code for the method, using the operations provided by the Sorted List ADT.
    c. Write a paragraph comparing the client method and the ADT method (Exercise 22) for the same task.

24. The algorithm for the Sorted List ADT insert operation starts at the beginning of the list and looks at each item, to determine where the insertion should take place. Once the insertion location is determined, the algorithm moves each list item between that location and the end of the list, starting at the end of the list, over to the next position. This creates space for the new item to be inserted. Another approach to this algorithm is just to start at the last location, examine the item there to see if the new item should be placed before it or after it, and shift the item in that location to the next location if the answer is "before." Repeating this procedure with the next to last item, then the one next to that, and so on, will eventually move all the items that need to be moved, so that when the answer is finally "after" (or the beginning of the list is reached) the needed location is available for the new item.
    a. Formalize this new algorithm with a pseudocode description, such as the algorithms presented in the text.
    b. Rewrite the `insert` method of the `SortedStringList` class to use the new algorithm.
    c. Test the new method.
25. The specifications for the Sorted List ADT `delete` operation state that the item to be deleted is on the list.
    a. Create a specification for a new form of delete, called `tryDelete`, that leaves the list unchanged if the item to be deleted is not in the list. The new delete operation should return a `boolean` value `true` if the item was found and deleted, `false` if the item was not on the list.
    b. Implement `tryDelete` as specified in (a).
26. The Sorted List ADT (for `SortedStringList`) is to be extended with an operation `merge`, which adds the contents of a list parameter to the current list.
    a. Write the specifications for this operation. The signature for the routine should be

       ```
       void merge(SortedStringList list)
       ```

    b. Design an algorithm for this operation.
    c. Devise a test plan for your algorithm.
    d. Implement and test your algorithm.
27. A String List ADT is to be extended by the addition of method `trimList`, which has the following specifications:

trimList(String lower, String upper)

Effect:	Removes all elements from the list that are less than `lower` and greater than `upper`
Postconditions:	This list contains only items that are between `lower` and `upper` inclusive

a. Implement `trimList` as a method of `UnsortedStringList`.
b. Implement `trimList` as a member method of `SortedStringList`.
c. Compare the algorithms used in (a) and (b).
d. Implement `trimList` as a client method of `UnsortedStringList`.
e. Implement `trimList` as a client method of `SortedStringList`.

3.5 Comparison of Algorithms

28. Describe the order of magnitude of each of the following functions using Big-O notation:
   a. $N^2 + 3N$
   b. $3N^2 + N$
   c. $N^5 + 100N^3 + 245$
   d. $3N\log_2 N + N^2$
   e. $1 + N + N^2 + N^3 + N^4$
   f. $(N * (N - 1)) / 2$
29. Give an example of an algorithm (other than the examples discussed in the chapter) that is
   a. $O(1)$
   b. $O(N)$
   c. $O(N^2)$
30. Describe the order of magnitude of each of the following code sections using Big-O notation:
   a.
```
count = 0;
for (i = 1; i <= N; i++)
  count++;
```
   b.
```
count = 0;
 for (i = 1; i <= N; i++)
   for (j = 1; j <= N; j++)
     count++;
```
   c.
```
value = N;
count = 0;
while (value > 1)
{
  value = value / 2;
  count++;
}
```
31. Algorithm 1 does a particular task in a "time" of N^3, where N is the number of elements processed. Algorithm 2 does the same task in a "time" of $3N + 1000$.
   a. What are the Big-O requirements of each algorithm?

b. Which algorithm is more efficient by Big-O standards?

c. Under what conditions, if any, would the "less efficient" algorithm execute more quickly than the "more efficient" algorithm?

3.6 Comparison of Unsorted and Sorted List ADT Algorithms

32. Assume that for each of the listed exercises an optimal algorithm was written (optimal means that it is not possible under the circumstances to write a faster algorithm). Give a Big-O estimate of the run time for the corresponding algorithms. Unless otherwise stated, let *N* represent the size of the list.

   a. Exercise 4a: `printList` for `UnsortedStringList`

   b. Exercise 5a: `compareLists` for `UnsortedStringList` (*N* = size of the larger list)

   c. Exercise 10: `isEmpty` for `UnsortedStringList`

   d. Exercise 11: `smallest` for `UnsortedStringList`

   e. Exercise 12: `smallest` for `UnsortedStringList` client

   f. Exercise 13: `tryDelete` for `UnsortedStringList`

   g. Exercise 22: `smallest` for `SortedStringList`

   h. Exercise 23: `smallest` for `SortedStringList` client

3.7 Generic ADTs

33. We did not devise a test plan for the `SortedList` class.

   a. Create an appropriate test plan using the `ListString` class to provide objects for storing on the list. Remember to include tests of the `retrieve` operation.

   b. Create a set of test input files that represents the completed test plan.

   c. Use the `TDSortedList` program to run and verify your tests.

34. Create a new concrete class, `UnsortedList`, that extends the `List` class, as discussed at the end of the section, A Generic Sorted List ADT.

35. Consider a `ListNumber` class that implements the `Listable` interface. The class defines two instance variables, one of primitive type `int` and the other of type `String`. The former acts as the key. The idea is that objects of the class can hold an integer value, for example, 5, and the corresponding string, "five". The class exports a constructor that accepts two parameters that are used to initialize the hidden instance variables, two observer methods that return the values of the hidden instance variables, a `toString` method that returns `value`, and of course the required `compareTo` and `copy` methods.

   a. Create the `ListNumber` class.

   b. Test your `ListNumber` class by using it with the `SortedList` class.

Case Study: Real Estate Listings

36. Devise and perform a thorough test of the Real Estate application program.

37. Explain how you would have to change the Real Estate program to handle each of the following specification changes. For each case, indicate which program units need to be changed and a general description of how the change could be implemented.

    a. The `houses.dat` file is redesigned to include the owner's first name first, and last name second, instead of vice versa.

    b. In the interface "Lot numbers" are to be referred to as "Locations".

    c. The information for each house is augmented by a "Number of bathrooms" attribute.

    d. In a surprising and unconventional move, the company decides that each house will have a unique price, and that houses should be listed in order of price instead of lot numbers.

38. Expand the Real Estate program so that the "blank label" field of the interface is used to always show the total number of houses on the list.

39. Expand the Real Estate program to include two more user interface buttons: largest and smallest. If the list of houses is empty and the user clicks on either of the new buttons, the message "List is empty" should appear in the status label area. Otherwise, when the user clicks on the "largest" button, the program should display the house information for the largest house in terms of square feet; and when the user clicks on the "smallest" button, the program should display the house information for the smallest house in terms of square feet.