CHAPTER 4

Data Abstraction: The Walls

This chapter elaborates on data abstraction, which was introduced in Chapter 2 as a technique for increasing the modularity of a program—for building "walls" between a program and its data structures. During the design of a solution, you will discover that you need to support several operations on the data and therefore need to define abstract data types (ADTs). This chapter introduces some simple abstract data types and uses them to demonstrate the advantages of abstract data types in general. In Part Two of this book, you will see several other important ADTs.

Only after you have clearly specified the operations of an abstract data type should you consider data structures for implementing it. This chapter explores implementation issues and introduces Java classes as a way to hide the implementation of an ADT from its users.

4.1 Abstract Data Types

Modularity is a technique that keeps the complexity of a large program manageable by systematically controlling the interaction of its components. You can focus on one task at a time in a modular program without other distractions. Thus, a modular program is easier to write, read, and modify. Modularity also isolates errors and eliminates redundancies.

A modular program is easier to write, read, and modify

You can develop modular programs by piecing together existing software components with methods that have yet to be written. In doing so, you should focus on *what* a module does and not on *how* it does it. To use existing software, you need a clear set of specifications that details how the modules behave. To write new methods, you need to decide what you would like them to do and proceed under the assumption that they exist and work. In this way you can write the methods in relative isolation from one another, knowing what each one will do but not necessarily *how* each will eventually do it. That is, you should practice **procedural abstraction.**

Write specifications for each module before implementing it

While writing a module's specifications, you must identify details that you can hide within the module. The principle of **information hiding** involves not only hiding these details, but also making them *inaccessible* from outside a module. One way to understand information hiding is to imagine **walls** around the various tasks a program performs. These walls prevent the tasks from becoming entangled. The wall around each task T prevents the other tasks from "seeing" how T is performed. Thus, if task Q uses task T, and if the method for performing task T changes, task Q will not be affected. As Figure 4-1 illustrates, the wall prevents task Q's method of solution from depending on task T's method of solution.

Isolate the implementation details of a module from other modules

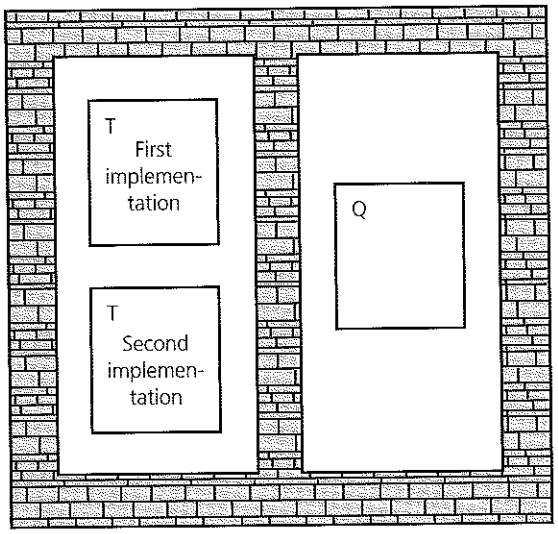

FIGURE 4-1

Isolated tasks: the implementation of task T does not affect task Q

The isolation of the modules cannot be total, however. Although task Q does not know *how* task T is performed, it must know *what* task T is and how to initiate it. For example, suppose that a program needs to operate on a sorted array of names. The program may, for instance, need to search the array for a given name or display the names in alphabetical order. The program thus needs a method S that sorts an array of names. Although the rest of the program knows that method S will sort an array, it should not care how S accomplishes its task. Thus, imagine a tiny slit in each wall, as Figure 4-2 illustrates. The slit is not large enough to allow the outside world to see the method's inner workings, but things can pass through the slit into and out of the method. For example, you can pass the array into the sort method, and the method can pass the sorted array out to you. What goes in and comes out is governed by the terms of the method's specifications, or **contract:** *If you use the method in this way, this is exactly what it will do for you.*

Often the solution to a problem requires operations on data. Such operations are broadly described in one of three ways:

Typical operations on data

- **Add** data to a data collection.
- **Remove** data from a data collection.
- **Ask questions** about the data in a data collection.

The details of the operations, of course, vary from application to application, but the overall theme is the management of data. Realize, however, that not all problems use or require these operations.

Both procedural and data abstraction ask you to think "what," not "how"

Data abstraction asks that you think in terms of *what* you can do to a collection of data independently of *how* you do it. Data abstraction is a technique that allows you to develop each data structure in relative isolation from the rest of the solution. The other modules of the solution will "know" what operations they can perform on the data, but they should not depend on how the data is stored or how the operations are performed. Again, the terms of the

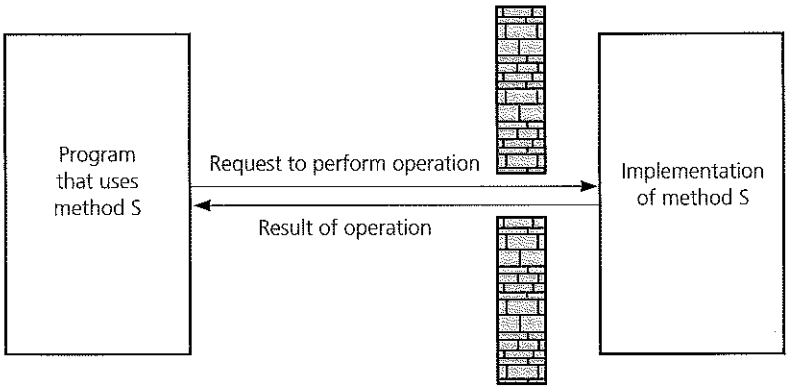

FIGURE 4-2

A slit in the wall

contract are *what* and not *how*. Thus, data abstraction is a natural extension of procedural abstraction.

An ADT is a collection of data and a set of operations on that data

A collection of data together with a set of operations on that data are called an **abstract data type,** or **ADT.** For example, suppose that you need to store a collection of names in a manner that allows you to search rapidly for a given name. The binary search algorithm described in Chapter 3 enables you to search an array efficiently, if the array is sorted. Thus, one solution to this problem is to store the names sorted in an array and to use a binary search algorithm to search the array for a specified name. You can view the *sorted array together with the binary search algorithm* as an ADT that solves this problem.

Specifications indicate what ADT operations do, but not how to implement them

The description of an ADT's operations must be rigorous enough to specify completely their effect on the data, yet it must not specify how to store the data nor how to carry out the operations. For example, the ADT operations should not specify whether to store the data in consecutive memory locations or in disjoint memory locations. You choose a particular **data structure** when you **implement** an ADT.

Data structures are part of an ADT's implementation

Recall that a data structure is a construct that you can define within a programming language to store a collection of data. For example, arrays, which are built into Java, are data structures. However, you can invent other data structures. For example, suppose that you wanted a data structure to store both the names and salaries of a group of employees. You could use the following Java statements:

```
final int MAX_NUMBER = 500;
String[] names = new String[MAX_NUMBER];
double[] salaries = new double[MAX_NUMBER];
```

Here the employee `names[i]` has a salary of `salaries[i]`. The two arrays `names` and `salaries` together form a data structure, yet Java has no single data type to describe it.

Carefully specify an ADT's operations before you implement them

When a program must perform data operations that are not directly supported by the language, you should first design an abstract data type and carefully specify what the ADT operations are to do (the contract). Then—*and only then*—should you implement the operations with a data structure. If you implement the operations properly, the rest of the program will be able to assume that the operations perform as specified—that is, that the terms of the contract are honored. However, the program must not depend on a particular approach for supporting the operations.

An abstract data type is not another name for a data structure.

ADTs and data structures are not the same

To give you a better idea of the conceptual difference between an ADT and a data structure, consider a refrigerator's ice dispenser, as Figure 4-3 illustrates. It has water as input and produces as output either chilled water, crushed ice, or ice cubes, according to which one of the three buttons you push. It also has an indicator that lights when no ice is presently available. The dispenser is analogous to an abstract data type. The water is analogous to data; the operations are *chill*, *crush*, *cube*, and *isEmpty*. At this level of design, you are not concerned with how the dispenser will perform its operations, only that it

KEY CONCEPTS

ADTs Versus Data Structures

- An abstract data type is a collection of data and a set of operations on that data.
- A data structure is a construct within a programming language that stores a collection of data.

performs them. If you want crushed ice, do you really care how the dispenser accomplishes its task as long as it does so correctly? Thus, after you have specified the dispenser's methods, you can design many uses for crushed ice without knowing how the dispenser accomplishes its tasks and without the distraction of engineering details.

Eventually, however, someone must build the dispenser. Exactly how will this machine produce crushed ice, for example? It could first make ice cubes and then either crush them between two steel rollers or smash them into small pieces by using hammers. Many other techniques are possible. The internal structure of the dispenser corresponds to the implementation of the ADT in a programming language—that is, to a data structure.

Although the owner of the dispenser does not care about its inner workings, he or she does want a design that is as efficient in its operation as possible. Similarly, the dispenser's manufacturer wants a design that is as easy and cheap to build as possible. You should have these same concerns when you choose a data structure to implement an ADT in Java. Even if you do not implement the ADT yourself, but instead use an already implemented ADT, you—like the person who buys a refrigerator—should care at least about the ADT's efficiency.

Notice that the dispenser is surrounded by steel walls. The only breaks in the walls accommodate the input (water) to the machine and its output (chilled water, crushed ice, or ice cubes). Thus, the machine's interior mechanisms are not only hidden from the user but also are inaccessible. In addition, the mechanism of one operation is hidden from and inaccessible to another operation.

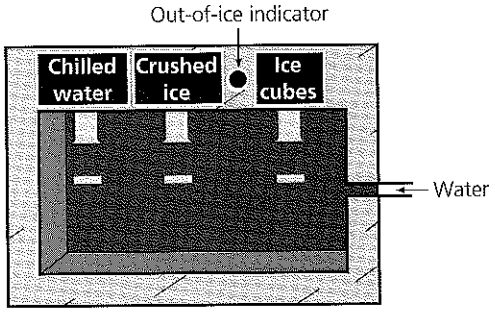

FIGURE 4-3

A dispenser of chilled water, crushed ice, and ice cubes

This modular design has benefits. For example, you can improve the operation *crush* by modifying its module without affecting the other modules. You could also add an operation by adding another module to the machine without affecting the original three operations. Thus, both abstraction and information hiding are at work here.

A program should not depend on the details of an ADT's implementation

To summarize, data abstraction results in a wall of ADT operations between data structures and the program that accesses the data within these data structures, as Figure 4-4 illustrates. If you are on the program's side of the wall, you will see an **interface** that enables you to communicate with the data structure. That is, you request the ADT operations to manipulate the data in the data structure, and they pass the results of these manipulations back to you.

Using an ADT is like using a vending machine

This process is analogous to using a vending machine. You press buttons to communicate with the machine and obtain something in return. The machine's external design dictates how you use it, much as an ADT's specifications govern what its operations are and what they do. As long as you use a vending machine according to its design, you can ignore its inner technology. As long as you agree to access data only by using ADT operations, your program can be oblivious to any change in the data structures that implement the ADT.

The following pages describe how to use an abstract data type to realize data abstraction's goal of separating the operations on data from the implementation of these operations. In doing so, we will look at several examples of ADTs.

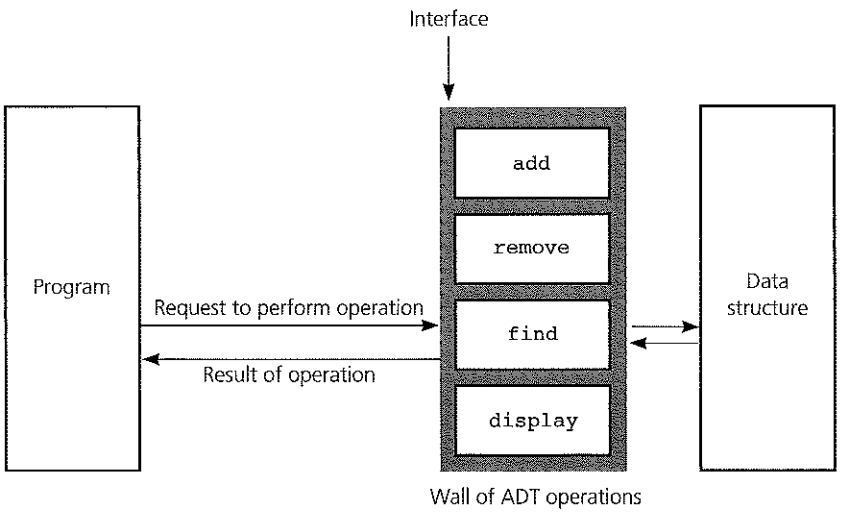

FIGURE 4-4

A wall of ADT operations isolates a data structure from the program that uses it

4.2 Specifying ADTs

To elaborate on the notion of an abstract data type, consider a list that you might encounter, such as a list of chores, a list of important dates, a list of addresses, or the grocery list pictured in Figure 4-5. As you write a grocery list, where do you put new items? Assuming that you write a neat one-column list, you probably add new items to the end of the list. You could just as well add items to the beginning of the list or add them so that your list is sorted alphabetically. Regardless, the items on a list appear in a sequence. The list has one first item and one last item. Except for the first and last items, each item has a unique **predecessor** and a unique **successor.** The first item—the **head** or **front** of the list—does not have a predecessor, and the last item—the **tail** or **end** of the list—does not have a successor.

Lists contain items of the same type: You can have a list of grocery items or a list of phone numbers. What can you do to the items on a list? You might count the items to determine the length of the list, add an item to the list, remove an item from the list, or look at **(retrieve)** an item. The items on a list, together with operations that you can perform on the items, form an abstract data type. You must specify the behavior of the ADT's operations on its data, that is, the list items. It is important that you focus only on specifying the operations and not on how you will implement them. In other words, do not bring to this discussion any preconceived notion of a data structure that the term "list" might suggest.

Where do you add a new item and which item do you want to look at? The various answers to these questions lead to several kinds of lists. You might decide to add, delete, and retrieve items only at the end of the list or only at the front of the list or at both the front and end of the list. The specifications of these lists are left as an exercise; next we will discuss a more general list.

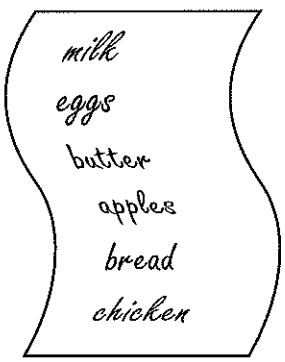

FIGURE 4-5

A grocery list

The ADT List

Once again, consider the grocery list pictured in Figure 4-5. The previously described lists, which manipulate items at one or both ends of the list, are not really adequate for an actual grocery list. You would probably want to access items anywhere on the list. That is, you might look at the item at position *i*, delete the item at position *i*, or insert an item at position *i* on the list. Such operations are part of the ADT **list.**

Note that it is customary to include an initialization operation that creates an empty list. Other operations that determine whether the list is empty or the length of the list are also useful.

You reference list items by their position within the list

Although the six items on the list in Figure 4-5 have a sequential order, they are not necessarily sorted by name. Perhaps the items appear in the order in which they occur on the grocer's shelves, but more likely they appear in the order in which they occurred to you as you wrote the list. The ADT list is simply an ordered collection of items that you reference by position number.

KEY CONCEPTS

ADT List Operations

1. Create an empty list.
2. Determine whether a list is empty.
3. Determine the number of items on a list.
4. Add an item at a given position in the list.
5. Remove the item at a given position in the list.
6. Remove all the items from the list.
7. Retrieve (get) the item at a given position in the list.

The following pseudocode specifies the operations for the ADT list in more detail. Figure 4-6 shows the UML diagram for this ADT.

To get a more precise idea of how the operations work, apply them to the grocery list

milk, eggs, butter, apples, bread, chicken

where milk is the first item on the list and chicken is the last item. To begin, consider how you can construct this list by using the operations of the ADT list. One way is first to create an empty list `aList` and then use a series of insertion operations to append successively the items to the list as follows:

```
aList.createList()
aList.add(1, milk)
aList.add(2, eggs)
```

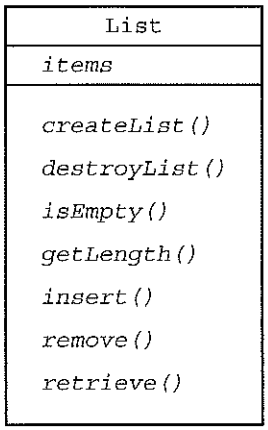

FIGURE 4-6

UML diagram for ADT `List`

```
aList.add(3, butter)
aList.add(4, apple)
aList.add(5, bread)
aList.add(6, chicken)
```

The notation[1] `aList.O` indicates that an operation `O` applies to the list `aList`.

Notice that the list's insertion operation can insert new items into any position of the list, not just at its front or end. According to `add`'s specification, if a new item is inserted into position *i*, the position of each item that was at a position of *i* or greater is increased by 1. Thus, for example, if you start with the previous grocery list and you perform the operation

```
aList.add(4, nuts)
```

the list `aList` becomes

milk, eggs, butter, nuts, apples, bread, chicken

All items that had position numbers greater than or equal to 4 before the insertion now have their position numbers increased by 1 after the insertion.

Similarly, the deletion operation specifies that if an item is deleted from position *i*, the position of each item that was at a position greater than *i* is decreased by 1. Thus, for example, if `aList` is the list

milk, eggs, butter, nuts, apples, bread, chicken

1. This notation is similar to the Java implementation of the ADT.

KEY CONCEPTS

Pseudocode for the ADT List Operations

```
+createList()
// Creates an empty list.

+isEmpty():boolean {query}
// Determines whether a list is empty.

+size():integer {query}
// Returns the number of items that are in a list.

+add(in index:integer, in item:ListItemType)
// Inserts item at position index of a list, if
// 1 <= index <= size()+1.
// If index <= size(), items are renumbered as
// follows: The item at index becomes the item at
// index+1, the item at index+1 becomes the
// item at index+2, and so on.
// Throws an exception when index is out of range or if
// the item cannot be placed on the list (list full).

+remove(in index:integer)
// Removes the item at position index of a list, if
// 1 <= index <= size(). If index < size(), items are
// renumbered as follows: The item at index+1 becomes
// the item at index, the item at index+2 becomes the
// item at index+1, and so on.
// Throws an exception when index is out of range or if
// the list is empty.

+removeAll()
// Removes all the items in the list.

+get(index):ListItemType {query}
// Returns the item at position index of a list
// if 1 <= index <= size(). The list is
// left unchanged by this operation.
// Throws an exception if index is out of range.
```

and you perform the operation

```
aList.remove(5)
```

the list becomes

milk, eggs, butter, nuts, bread, chicken

All items that had position numbers greater than 5 before the deletion now have their position numbers decreased by 1 after the deletion.

These examples illustrate that an ADT can specify the effects of its operations without having to indicate how to store the data. The specifications of the seven operations are the sole terms of the contract for the ADT list: *If you request that these operations be performed, this is what will happen.* The specifications contain no mention of how to store the list or how to perform the operations; they tell you only what you can do to the list. It is of fundamental importance that the specification of an ADT *not* include implementation issues. This restriction on the specification of an ADT is what allows you to build a wall between an implementation of an ADT and the program that uses it. (Such a program is called a **client.**) The behavior of the operations is the only thing on which a program should depend.

An ADT specification should not include implementation issues

A program should depend only on the behavior of the ADT

Note that the insertion, deletion, and retrieval operations throw an exception when the argument `index` is out of range. This technique provides the ADT with a simple mechanism to communicate operation failure to its client. For example, if you try to delete the tenth item from a five-item list, `remove` can throw an exception indicating that `index` is out of range. Exceptions enable the client to handle error situations in an implementation-independent way.

What does the specification of the ADT list tell you about its behavior? It is apparent that the list operations fall into the three broad categories presented earlier in this chapter.

- The operation `add` **adds** data to a data collection.
- The operations `remove` and `removeAll` **remove** data from a data collection.
- The operations `isEmpty`, `size`, and `get` **ask questions** about the data in a data collection.

Once you have satisfactorily specified the behavior of an ADT, you can design applications that access and manipulate the ADT's data solely in terms of its operations and without regard for its implementation. As a simple example, suppose that you want to display the items in a list. Even though the wall between the implementation of the ADT list and the rest of the program prevents you from knowing how the list is stored, you can write a method `displayList` in terms of the operations that define the ADT list. The pseudocode for such a method follows:[2]

2. In this example, `displayList` is not an ADT operation, so a procedural notation that specifies `aList` as a parameter is used.

An implementation-independent application of the ADT list

```
displayList(in aList:List)
// Displays the items on the list aList.

   for (index = 1 through aList.size()) {
      dataItem = aList.get(index)
      Display dataItem
   }  // end for
```

Notice that as long as the ADT list is implemented correctly, the `displayList` method will perform its task. In this case, `get` successfully retrieves each list item, because `index`'s value is always valid.

The method `displayList` does not depend on *how* you implement the list. That is, the method will work regardless of whether you use an array or some other data structure to store the list's data. This feature is a definite advantage of abstract data types. In addition, by thinking in terms of the available ADT operations, you will not be distracted by implementation details. Figure 4-7 illustrates the wall between `displayList` and the implementation of the ADT list.

As another application of the ADT operations, suppose that you want a method `replace` that replaces the item in position i with a new item. If the i^{th}

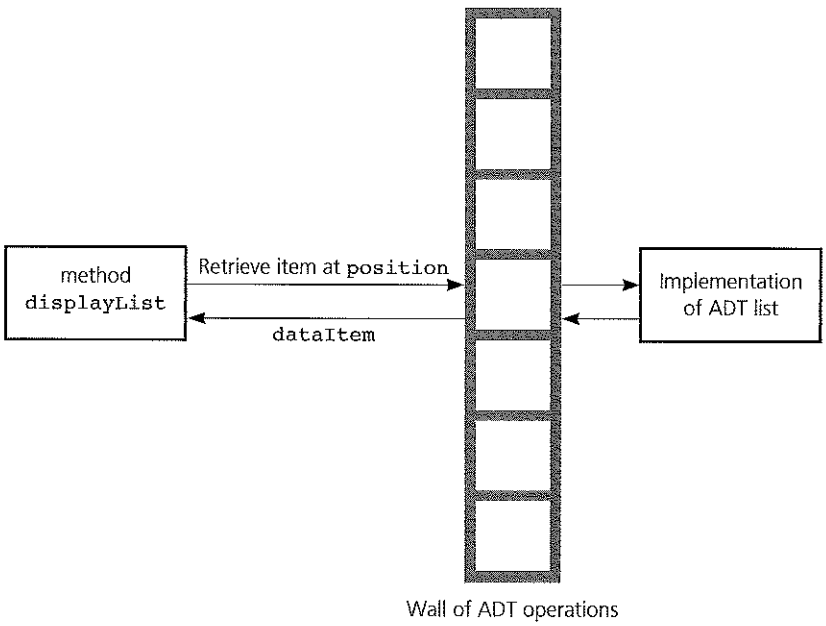

FIGURE 4-7

The wall between **`displayList`** and the implementation of the ADT list

item exists, *replace* deletes the item and inserts the new item at position *i*, as follows:

```
replace(in aList:List, in i:integer,
        in newItem:ListItemType
// Replaces the ith item on the list aList with
// newItem.

   if (i >=1 and i <= aList.size()) {
      aList.remove(i)
      aList.add(i, newItem)
   }  // end if
```

You can use ADT operations in an application without the distraction of implementation details

In both of the preceding examples, notice how you can focus on the task at hand without the distraction of implementation details such as arrays. With less to worry about, you are less likely to make an error in your logic when you use the ADT operations in applications such as *displayList* and *replace*. Likewise, when you finally implement the ADT operations in Java, you will not be distracted by these applications. In addition, because *displayList* and *replace* do not depend on any implementation decisions that you make for the ADT list, they are not altered by your decisions. These assertions assume that you do not change the specifications of the ADT operations when you implement them. However, as Chapter 2 pointed out, developing software is not a linear process. You may realize during implementation that you need to refine your specifications. Clearly, changes to the specification of any module affect any already-designed uses of that module.

To summarize, you can specify the behavior of an ADT independently of its implementation. Given such a specification, and without any knowledge of how the ADT will be implemented, you can design applications that use the ADT's operations to access its data.

The ADT Sorted List

One of the most frequently performed computing tasks is the maintenance, in some *specified* order, of a collection of data. Many examples immediately come to mind: students placed in order by their names, baseball players listed in order by their batting averages, and corporations listed in order by their assets. These orders are called **sorted.** In contrast, the items on a grocery list might be ordered—the order in which they appear on the grocer's shelves, for example—but they are probably not sorted by name.

The problem of *maintaining* sorted data requires more than simply sorting the data. Often you need to insert some new data item into its proper, sorted place. Similarly, you often need to delete some data item. For example, suppose your university maintains an alphabetical list of the students who are currently enrolled. The registrar must insert names into and delete names from this list because students constantly enroll in and leave school. These operations should preserve the sorted order of the data.

The ADT sorted list maintains items in sorted order

The following specifications define the operations for the ADT **sorted list.**

KEY CONCEPTS

Pseudocode for the ADT Sorted List Operations

```
+createSortedList()
// Creates an empty sorted list.

+sortedIsEmpty():boolean {query}
// Determines whether a sorted list is empty.

+sortedSize():integer {query}
// Returns the number of items that are in a sorted list.

+sortedAdd(in item:ListItemType)
// Inserts item into its proper sorted position in a
// sorted list. Throws an exception if the item
// cannot be placed on the list (list full).

+sortedRemove(in item:ListItemType)
// Deletes item from a sorted list.
// Throws an exception if the item is not found.

+sortedGet(in index:integer)
// Returns the item at position index of a
// sorted list, if 1 <= index <= sortedSize().
// The list is left unchanged by this operation.
// Throws an exception if the index is out of range.

+locateIndex(in item:ListItemType):integer {query}
// Returns the position where item belongs or
// exists in a sorted list; item and the list are
// unchanged.
```

The ADT sorted list differs from the ADT list in that a sorted list inserts and deletes items by their values and not by their positions. For example, `sortedAdd` determines the proper position for `item` according to its value. Also, `locateIndex`—which determines the position of any item, given its value—is a sorted list operation but not a list operation. However, `sortedGet` is like list's `get`: Both operations retrieve an item, given its position. The method `sortedGet` enables you, for example, to retrieve and then display each item in a sorted list.

Designing an ADT

The design of an abstract data type should evolve naturally during the problem-solving process. As an example of how this process might occur,

suppose that you want to determine the dates of all the holidays in a given year. One way to do this is to examine a calendar. That is, you could consider each day in the year and ascertain whether that day is a holiday. The following pseudocode is thus a possible solution to this problem:

```
listHolidays(in year:integer)
// Displays the dates of all holidays in a given year.

   date = date of first day of year
   while (date is before the first day of year+1) {
      if (date is a holiday) {
         write (date + " is a holiday")
      } // end if
      date = date of next day
   } // end while
```

What data is involved here? Clearly, this problem operates on dates, where a date consists of a month, day, and year. What operations will you need to solve the holiday problem? Your ADT must specify and restrict the legal operations on the dates just as the fundamental data type *int* restricts you to operations such as addition and comparison. You can see from the previous pseudocode that you must

What data does a problem require?

- Determine the date of the first day of a given year
- Determine whether a date is before another date
- Determine whether a date is a holiday
- Determine the date of the day that follows a given date

What operations does a problem require?

Thus, you could define the following operations for your ADT:

```
+firstDay(in year:integer):Date {query}
// Returns the date of the first day of a given year.

+isBefore(in date1:Date,
          in date2:Date) : Date {query}
// Returns true if date1 is before date2,
// otherwise returns false.

+isHoliday(in aDate:Date) : boolean {query}
// Returns true if date is a holiday,
// otherwise returns false.

+nextDay(in aDate:Date) : Date
// Returns the date of the day after a given date.
```

The *listHolidays* pseudocode now appears as follows:

```
listHolidays(in year:integer)
// Displays the dates of all holidays in a given year.

   date = firstDay(year)
   while (isBefore(date, firstDay(year+1))) {
      if (isHoliday(date)) {
         write (date + " is a holiday ")
      } // end if
      date = nextDay(date)
   } // end while
```

Thus, you can design an ADT by identifying data and choosing operations that are suitable to your problem. After specifying the operations, you use them to solve your problem independently of the implementation details of the ADT.

An appointment book.. As another example of an ADT design, imagine that you want to create a computerized appointment book that spans a one-year period. Suppose that you make appointments only on the hour and half hour between 8 A.M. and 5 P.M. For simplicity, assume that all appointments are 30 minutes in duration. You want your system to store a brief notation about the nature of each appointment along with the date and time.

To solve this problem, you can define an ADT appointment book. The data items in this ADT are the appointments, where an appointment consists of a date, time, and purpose. What are the operations? Two obvious operations are

- Make an appointment for a certain date, time, and purpose. (You will want to be careful that you do not make an appointment at an already occupied time.)
- Cancel the appointment for a certain date and time.

In addition to these operations, it is likely that you will want to

- Ask whether you have an appointment at a given time.
- Determine the nature of your appointment at a given time.

Finally, ADTs typically have initialization operations.

Thus, the ADT appointment book can have the following operations:

```
+createAppointmentBook()
// Creates an empty appointment book.

+isAppointment(in apptDate:Date,
               in apptTime:Time):boolean {query}
// Returns true if an appointment exists for the date
// and time specified; otherwise returns false.
```

```
+makeAppointment(in apptDate:Date, in apptTime:Time,
                 in purpose:string):boolean
// Inserts the appointment for the date, time, and purpose
// specified as long as it does not conflict with an
// existing appointment.
// Returns true if successful, false otherwise.

+cancelAppointment(in apptDate:Date,
                   in apptTime:Time):boolean
// Deletes the appointment for the date and time specified.
// Returns true if successful, false otherwise.

+checkAppointment(in apptDate:Date,
                  in apptTime:Time):string {query}
// Returns the purpose of the appointment at
// the given date/time, if one exists. Otherwise, returns
// null.
```

You can use these ADT operations to design other operations on the appointments. For example, suppose that you want to change the date or time of a particular appointment within the existing appointment book *apptBook*. The following pseudocode indicates how to accomplish this task by using the previous ADT operations:

```
// change the date or time of an appointment

read (oldDate, oldTime, newDate, newTime)
// get purpose of appointment
purpose = apptBook.checkAppointment(oldDate, oldTime)
if (purpose exists) {
  // see if new date/time is available
  if (apptBook.isAppointment(newDate, newTime)) {
    // new date/time is booked
    write ("You already have an appointment at " + newTime +
           " on " + newDate)
  }
  else  { // new date/time is available
    apptBook.cancelAppointment(oldDate, oldTime))
    if (apptBook.makeAppointment(newDate, newTime,
        purpose)){
      write ("Your appointment has been rescheduled to" +
             newTime + " on " + newDate)
    }  //end if
  }  // end if
}
else {
```

```
    write ("You do not have an appointment at " + oldTime +
           " on " + oldDate)
}  // end if
```

You can use an ADT without knowledge of its implementation

Again notice that you can design applications of ADT operations without knowing how the ADT is implemented. The exercises at the end of this chapter provide examples of other tasks that you can perform with this ADT.

ADTs that suggest other ADTs. . Both of the previous examples require you to represent a date; the appointment book example also requires you to represent the time. Java has a `java.util.Date` class that you can use to represent the date and time. You can also design ADTs to represent these items. It is not unusual for the design of one ADT to suggest other ADTs. In fact, you can use one ADT to implement another ADT. The programming problems at the end of this chapter ask you to design and implement the simple ADTs date and time.

You can use an ADT to implement another ADT

This final example also describes an ADT that suggests other ADTs for its implementation. Suppose that you want to design a database of recipes. You could think of this database as an ADT: The recipes are the data items, and some typical operations on the recipes could include the following:

```
+insertRecipe(in aRecipe:Recipe)
// Inserts recipe into the database.

+deleteRecipe(in aRecipe:Recipe)
// Deletes recipe from the database.

+retrieveRecipe(in name:string):Recipe {query}
// Retrieves the named recipe from the database.
```

This level of the design does not indicate such details as where `insertRecipe` will place a recipe into the database.

Now imagine that you want to write a method that scales a recipe retrieved from the database: If the recipe is for n people, you want to revise it so that it will serve m people. Suppose that the recipe contains measurements such as 2½ cups, 1 tablespoon, and ¼ teaspoon. That is, the quantities are given as mixed numbers—integers and fractions—in units of cups, tablespoons, and teaspoons.

This problem suggests another ADT—measurement—with the following operations:

```
+getMeasure():Measurement {query}
// Returns the measure.

+setMeasure(in m:Measurement)
// Sets the measure.
```

```
+scaleMeasure(in scaleFactor: float):Measurement
// Multiplies measure by a fractional scaleFactor, which
// has no units, and returns the result.

+convertMeasure(in oldUnits:MeasureUnit,
                in newUnits:MeasureUnit):Measurement {query}
// Converts measure from its old units to a measure in
// new units, and returns the result.
```

Suppose that you want the ADT measurement to perform exact fractional arithmetic. Because our planned implementation language Java does not have a data type for fractions and floating-point arithmetic is not exact, another ADT called fraction is in order. Its operations could include addition, subtraction, multiplication, and division of fractions. For example, you could specify addition as

```
addFractions(in first:Fraction, in second:Fraction):Fraction
// Adds two fractions and returns the sum reduced to lowest
// terms.
```

Moreover, you could include operations to convert a mixed number to a fraction and to convert a fraction to a mixed number when feasible.

When you finally implement the ADT measurement, you can use the ADT fraction. That is, you can use one ADT to implement another ADT.

Axioms (Optional)

The previous specifications for ADT operations have been stated rather informally. For example, they rely on your intuition to know the meaning of "an item is at position i" in an ADT list. This notion is simple, and most people will understand its intentions. However, some abstract data types are much more complex and less intuitive than a list. For such ADTs, you should use a more rigorous method of defining the behavior of their operations: You must supply a set of mathematical rules—called **axioms**—that precisely specify the behavior of each ADT operation.

An axiom is a mathematical rule

An axiom is actually an invariant—a true statement—for an ADT operation. For example, you are familiar with axioms for algebraic operations; in particular, you know the following rules for multiplication:

Axioms for multiplication

$$(a \times b) \times c = a \times (b \times c)$$

$$a \times b = b \times a$$

$$a \times 1 = a$$

$$a \times 0 = 0$$

These rules, or axioms, are true for any numeric values of a, b, and c, and they describe the behavior of the multiplication operator $\times$.

Axioms specify the behavior of an ADT

In a similar fashion, you can write a set of axioms that completely describes the behavior of the operations for the ADT list. For example,

A newly created list is empty

is an axiom since it is true for all newly created lists. You can state this axiom succinctly in terms of the operations of the ADT list as follows:

```
(aList.createList()).isEmpty() is true
```

That is, the list `aList` is empty.

The statement

If you insert an item x into the i^{th} position of an ADT list, retrieving the i^{th} item will result in x

is true for all lists, and so it is an axiom. You can state this axiom in terms of the operations of the ADT list as follows:[3]

```
(aList.add(i, x)).get(i) = x
```

That is, `get` retrieves from position `i` of list `aList` the item `x` that `add` has put there.

The following axioms formally define the ADT list:

KEY CONCEPTS

Axioms for the ADT List

1. `(aList.createList()).size() = 0`
2. `(aList.add(i, x)).size() = aList.size() + 1`
3. `(aList.remove(i)).size() = aList.size() - 1`
4. `(aList.createList()).isEmpty() = true`
5. `(aList.add(i, item)).isEmpty() = false`
6. `(aList.createList()).remove(i) = error`
7. `(aList.add(i, x)).remove(i) = aList`
8. `(aList.createList()).get(i) = error`
9. `(aList.add(i, x)).get(i) = x`
10. `aList.get(i) = (aList.add(i, x)).get(i+1)`
11. `aList.get(i+1) = (aList.remove(i)).get(i)`

3. The = notation within these axioms denotes algebraic equality.

A set of axioms does not make the pre- and postconditions for an ADT's operations unnecessary. For example, the previous axioms do not describe *add*'s behavior when you try to insert an item into position 50 of a list of 2 items. One way to handle this situation is to include the restriction

```
1 <= index <= size()+1
```

in *add*'s precondition. Another way—which you will see when we implement the ADT list later in this chapter—does not restrict *index*, but rather throws an exception if *index* is outside the previous range. Thus, you need both a set of axioms and a set of pre- and postconditions to define the behavior of an ADT's operations completely.

Use axioms to determine the effect of a sequence of ADT operations

You can use axioms to determine the outcome of a sequence of ADT operations. For example, if *aList* is a list of characters, how does the sequence of operations

```
aList.add(1, b)
aList.add(1, a)
```

affect *aList*? We will show that *a* is the first item in this list and that *b* is the second item by using *get* to retrieve these items.

You can write the previous sequence of operations in another way as

```
(aList.add(1, b)).add(1, a)
```

or

```
tempList.add(1, a)
```

where *tempList* represents *aList.add(1, b)*. Now retrieve the first and second items in the list *tempList.add(1, a)*, as follows:

```
(tempList.add(1, a)).get(1) = a
```

by axiom 9

and

```
(tempList.add(1, a)).get(2)
   = tempList.get(1)                  by axiom 10
   = (aList.add(1, b)).get(1)         by definition of tempList
   = b                                by axiom 9
```

Thus, *a* is the first item in the list and *b* is the second item.

Axioms are treated further in exercises in the rest of the book.

4.3 Implementing ADTs

The previous sections emphasized the specification of an abstract data type. When you design an ADT, you concentrate on what its operations do, but you ignore how you will implement them. The result should be a set of clearly specified ADT operations.

Data structures are part of an ADT's implementation

How do you implement an ADT once its operations are clearly specified? That is, how do you store the ADT's data and carry out its operations? Earlier in this chapter you learned that when implementing an ADT, you choose data structures to represent the ADT's data. Thus, your first reaction to the implementation question might be to choose a data structure and then to write methods that access it in accordance with the ADT operations. Although this point of view is not incorrect, hopefully you have learned not to jump right into code. In general, you should refine an ADT through successive levels of abstraction. That is, you should use a top-down approach to designing an algorithm for each of the ADT operations. You can view each of the successively more concrete descriptions of the ADT as implementing its more abstract predecessors. The refinement process stops when you reach data structures that are available in your programming language. The more primitive your language, the more levels of implementation you will require.

The choices that you make at each level of the implementation can affect its efficiency. For now, our analyses will be intuitive, but Chapter 10 will introduce you to quantitative techniques that you can use to weigh the trade-offs involved.

Recall that the program that uses the ADT should see only a wall of available operations that act on data. Figure 4-8 illustrates this wall once again. Both the data structure that you choose to contain the data and the implementations of the ADT operations are hidden behind the wall. By now, you should realize the advantage of this wall.

In a non-object-oriented implementation, both the data structure and the ADT operations are distinct pieces. The client agrees to honor the wall by

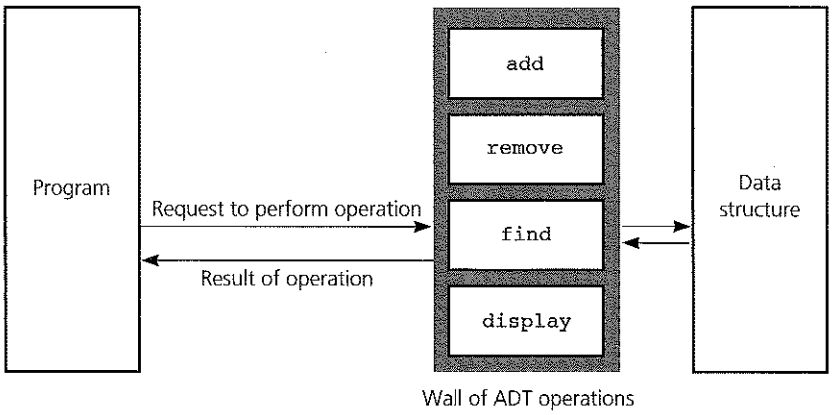

FIGURE 4-8

ADT operations provide access to a data structure

using only the ADT operations to access the data structure. Unfortunately, the data structure is hidden only if the client does not look over the wall! Thus, the client can violate the wall—either intentionally or accidentally—by accessing the data structure directly, as Figure 4-9 illustrates. Why is such an action undesirable? Later, this chapter will use an array `items` to store an ADT list's items. In a program that uses such a list, you might, for example, accidentally access the first item in the list by writing

```
firstItem = items[0];
```

instead of by invoking the list operation `get`. If you changed to another implementation of the list, your program would be incorrect. To correct your program, you would need to locate and change all occurrences of `items[0]`—but first you would have to realize that `items[0]` is in error!

Object-oriented languages such as Java provide a way for you to enforce the wall of an ADT, thereby preventing access of the data structure in any way other than by using the ADT operations. We will spend some time now exploring this aspect of Java by discussing classes, interfaces, and exceptions.

Java Classes Revisited

Recall from Chapter 2 that object-oriented programming, or OOP, views a program not as a sequence of actions but as a collection of components called objects. Encapsulation—one of OOP's three fundamental principles[4]—enables

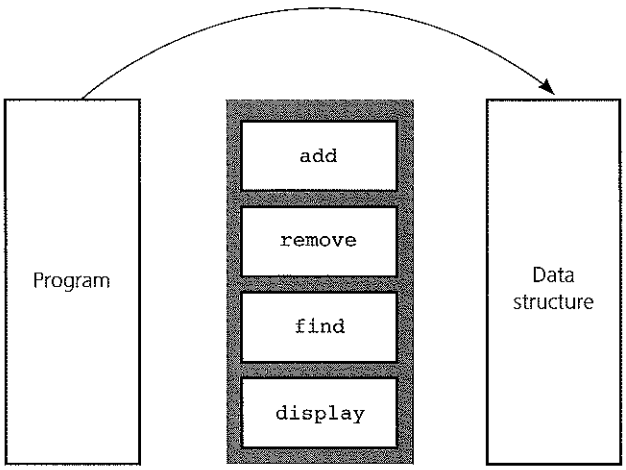

FIGURE 4-9

Violating the wall of ADT operations

4. The other principles are inheritance and polymorphism, which Chapter 9 will discuss.

you to enforce the walls of an ADT. It is, therefore, essential to an ADT's implementation and our main focus here.

Encapsulation combines an ADT's data with its operations—called **methods**—to form an **object.** Rather than thinking of the many components of the ADT in Figure 4-8, you can think at a higher level of abstraction when you consider the object in Figure 4-10 because it is a single entity. The object hides its inner detail from the programmer who uses it. Thus, an ADT's operations become an object's behaviors.

Encapsulation hides implementation details

We could use a ball as an example of an object. Because thinking of a basketball, volleyball, tennis ball, or soccer ball probably suggests images of the game rather than the object itself, let's abstract the notion of a ball by picturing a sphere. A sphere of a given radius has attributes such as volume and surface area. A sphere as an object should be able to report its radius, volume, surface area, and so on. That is, the sphere object has methods that return such values. This section will develop the notion of a sphere as an object. Later, in Chapter 9, you will see how to derive a ball from a sphere.

A Java class defines a new data type

In Java, a class is a new data type whose instances are objects. A class contains **data fields** and **methods,** collectively known as class members. Methods typically act on the data fields. By default, all members in a class are **private**—they are not directly accessible by your program—unless you designate them as **public.** The implementations of a class's methods, however, can use any private members of that class.

An object is an instance of a class

A class's data fields should be private

You should almost always declare a class's data fields as private. Typically, as was mentioned in Chapter 2, you provide methods—such as `setDataField` and `getDataField`—to access the data fields. In this way, you control how and whether the rest of the program can access the data fields. This design principle should lead to programs that not only are easier to debug, but also have fewer logical errors from the beginning.

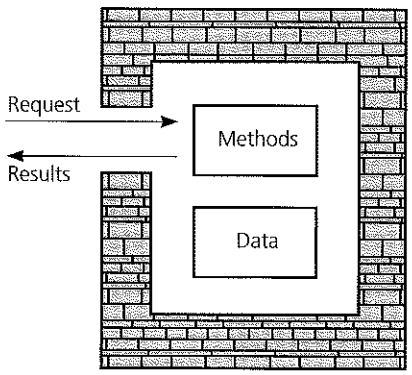

FIGURE 4-10

An object's data and methods are encapsulated

You should also distinguish between a class's data fields and any local variables that the implementation of a method requires. In Java, data fields are variables that are shared by all of the methods in the class. Data fields have initial default values, based on their type, and thus do not need to be explicitly initialized. But it is considered to be good programming practice to explicitly initialize data fields in the constructors when necessary. Local variables are used only within a single method and must be initialized explicitly before they are used.

The ADTs that you saw earlier had an operation for their creation. Classes have such methods, called **constructors.** A constructor creates and initializes new instances of a class. A typical class has several constructors. A constructor has the same name as the class. Constructors have no return type—not even *void*—and cannot use *return* to return a value. They can, however, have parameters. We will discuss constructors in more detail shortly, after we look at an example of a class definition.

A constructor creates and initializes an object

Java has a garbage collection mechanism to destroy objects that a program no longer needs. When a program no longer references an object, the Java runtime environment marks it for garbage collection. Periodically, the Java runtime environment executes a method that returns the memory used by these marked objects to the system for future use. Sometimes when an object is destroyed, other tasks beyond memory deallocation are necessary. In these cases, you define a *finalize* method for the object.

Java destroys objects that a program no longer references

Other features of classes that will be useful throughout this text include the ability to display objects and to compare two objects. To facilitate the discussion of constructors and these other features, consider a class of sphere objects called *Sphere*:

```
public class Sphere {
  private double theRadius;

  public Sphere() {
  // Default constructor: Creates a sphere and
  // initializes its radius to a default value.
  // Precondition: None.
  // Postcondition: A sphere of radius 1 exists.
    setRadius(1.0);
  } // end default constructor

  public Sphere(double initialRadius) {
  // Constructor: Creates a sphere and initializes
  // its radius.
  // Precondition: initialRadius is the desired radius.
  // Postcondition: A sphere of radius initialRadius
  // exists.
    setRadius(initialRadius);
  } // end constructor

  public void setRadius(double newRadius) {
```

```
// Sets (alters) the radius of an existing sphere.
// Precondition: newRadius is the desired radius.
// Postcondition: The sphere's radius is newRadius.
  if (newRadius >= 0.0) {
    theRadius = newRadius;
  }  // end if
} // end setRadius

public double radius() {
// Determines a sphere's radius.
// Precondition: None.
// Postcondition: Returns the radius.
  return theRadius;
} // end radius

public double diameter() {
// Determines a sphere's diameter.
// Precondition: None.
// Postcondition: Returns the diameter.
  return 2.0 * theRadius;
} // end diameter

public double circumference() {
// Determines a sphere's circumference.
// Precondition: None.
// Postcondition: Returns the circumference.
  return Math.PI * diameter();
} // end circumference

public double area() {
// Determines a sphere's surface area.
// Precondition: None.
// Postcondition: Returns the surface area.
  return 4.0 * Math.PI * theRadius * theRadius;
} // end area

public double volume() {
// Determines a sphere's volume.
// Precondition: None.
// Postcondition: Returns the volume.
  return (4.0*Math.PI * Math.pow(theRadius, 3.0)) / 3.0;
} // end volume

public void displayStatistics() {
// Displays statistics of a sphere.
// Precondition: Assumes System.out is available.
// Postcondition: None.
```

```
    System.out.println("\nRadius = " + radius() +
                       "\nDiameter = " + diameter() +
                       "\nCircumference = " + circumference() +
                       "\nArea = " + area() +
                       "\nVolume = " + volume());
  }  // end displayStatistics
} // end Sphere
```

Constructors. A constructor allocates memory for an object and can initialize the object's data to particular values. A class can have more than one constructor, as is the case for the class *Sphere*.

A default constructor has no parameters

The first constructor in *Sphere* is the **default constructor.** A default constructor by definition has no parameters. Typically, a default constructor initializes data fields to values that the class implementation chooses. For example, the implementation

```
public Sphere() {
   setRadius(1.0);
} // end default constructor
```

sets *theRadius* to 1.0. The following statement invokes the default constructor, which creates the object *unitSphere* and sets its radius to 1.0:

```
Sphere unitSphere = new Sphere();
```

The next constructor in *Sphere* is

```
public Sphere(double initialRadius) {
   setRadius(initialRadius);
} // end constructor
```

It creates a sphere object of radius *initialRadius*. You invoke this constructor by writing a declaration such as

```
Sphere mySphere = new Sphere(5.1);
```

In this case, the object *mySphere* has a radius of 5.1.

If you omit all constructors from your class, the compiler will generate a default constructor—that is, one with no parameters—for you. A **compiler-generated default constructor,** however, might not initialize data fields to values that you will find suitable.

If you define a constructor that has parameters, but you omit the default constructor, the compiler will not generate one for you. Thus, you will not be able to write statements such as

```
Sphere defaultSphere = new Sphere();
```

Using *Sphere*. The following simple program demonstrates the use of the class *Sphere*:

```
public static void main(String args[]) {
   // unitSphere radius is 1.0
   Sphere unitSphere = new Sphere();
   // mySphere radius is 5.1
   Sphere mySphere = new Sphere(5.1);

   unitSphere.displayStatistics();
   mySphere.setRadius(4.2);    // resets radius to 4.2
   System.out.println(mySphere.diameter());
}  // end main
```

A reference to the private data field ***theRadius*** would be illegal within this program

An object such as *mySphere* can, upon request, reset the value of its radius; return its radius; compute its diameter, surface area, circumference, and volume; and display these statistics. These requests to an object are called **messages** and are simply calls to methods. Thus, an object responds to a message by acting on its data. To invoke an object's method, you qualify the method's name—such as *setRadius*—with the object variable—such as *mySphere*.

The previous program is an example of a **client** of a class. A client of a particular class is simply a program or module that uses the class. We will reserve the term **user** for the person who uses a program.

Inheritance. A brief discussion of inheritance is provided here, since it is a common way to create new classes in Java. A more complete discussion of inheritance appears in Chapter 9.

Suppose that we want to create a class for colored spheres, knowing that we have already developed the class *Sphere*. We could write an entirely new class for the colored spheres, but if the colored spheres are actually like the spheres in the class *Sphere*, we can reuse the *Sphere* implementation and add color operations and characteristics by using inheritance. Here is an implementation of the class *ColoredSphere* that uses inheritance:

A class derived from the class ***Sphere***

```
import java.awt.Color;
public class ColoredSphere extends Sphere {
  private Color color;

  public ColoredSphere(Color c) {
  // Constructor: Creates a sphere and initializes its
  // radius to 1.
  // Precondition: c is the desired color.
  // Postcondition: A sphere of radius 1.0
  // and color c exists.
    super();
    color = c;
  }  // end constructor
```

```
  public ColoredSphere(Color c, double initialRadius) {
  // Constructor: Creates a sphere and initializes its
  // radius.
  // Precondition: initialRadius is the desired radius,
  // c is the desired color.
  // Postcondition: A sphere of radius initialRadius
  // and color c exists.

    super(initialRadius);
    color = c;
  } // end constructor

  public void setColor(Color c) {
  // Sets the color of the sphere.
  // Precondition: c is the desired color.
  // Postcondition: The sphere color is now c.
    color = c;
  }  // end setColor

  public Color getColor() {
  // Returns the color of the sphere.
  // Precondition: c is the desired color.
  // Postcondition: None.
    return color;
  }  // end getColor
}  // end ColoredSphere
```

Sphere is called the **base class** or **superclass,** and *ColoredSphere* is called the **derived class** or **subclass** of the class *Sphere*. The definition of the subclass includes an *extends* clause that indicates the superclass to be used. When you declare a class without an *extends* clause, you are implicitly extending the class *Object*, so *Object* is its superclass.

The subclass inherits the contents of the superclass, details of which are discussed in Chapter 9. For the moment, suffice it to say that the subclass will have all of the public members of the superclass available. Any instance of the subclass is also considered to be an instance of the superclass and can be used in a program anywhere that an instance of the superclass can be used. Also, any of the publicly defined methods or variables that can be used with instances of the superclass can be used with instances of the subclass. The subclass instances also have the methods and variables that are publicly defined in the subclass definition.

Public members of the superclass are available in the subclass

In the constructor for the *ColoredSphere* class, notice the use of the keyword **super**. You use this keyword to call the constructor of the superclass, so *super()* calls the constructor *Sphere()*, and *super(initialRadius)* calls the constructor *Sphere(double initialRadius)*. If the subclass constructor explicitly calls the superclass constructor, the call to *super* must precede all

A constructor in a subclass should invoke ***super*** to call the constructor of the superclass

other statements in the subclass constructor. Note that if a subclass constructor contains no call to the superclass constructor, the default superclass constructor is implicitly called.

Here is a method that uses the *ColoredSphere* class:

An instance of a subclass can invoke public methods of the superclass

```
public void useColoredSphere() {
  ColoredSphere ball =
        new ColoredSphere(java.awt.Color.white);
  ball.setRadius(5.0);
  System.out.println("The ball diameter is " +
                     ball.diameter());
  System.out.println("The ball color is " +
                     ball.getColor());
  // other code here...
}  // end useColorSphere
```

This method uses the constructor and the method *getColor* from the subclass *ColoredSphere*. It also uses the methods *setRadius* and *diameter* that are defined in the superclass *Sphere*.

Object equality.. Another task that can be useful in program development is to determine whether two objects are equal to one another. The *Object* class provides a method *equals* that compares two objects and returns *true* if they are actually the same object. For example,

Default ***equals*** as defined in the class ***Object*** compares two references

```
Sphere sphere1 = new Sphere();
Sphere sphere2 = sphere1;
if (sphere1.equals(sphere2)) {
  System.out.println("sphere1 and sphere2 are the " +
                     "same object");
}
else {
  System.out.println("sphere1 and sphere2 are " +
                     "different objects");
}  // end if
```

will produce the output

```
sphere1 and sphere2 are the same object
```

since both *sphere1* and *sphere2* reference the same object.

Suppose, however, that you want to determine whether two spheres have the same radius. For example,

```
Sphere sphere1 = new Sphere(2.0);
Sphere sphere3 = new Sphere(2.0);
if (sphere1.equals(sphere3)) {
```

```
  System.out.println("sphere1 and sphere3 have " +
                     "the same radius");
}
else {
  System.out.println("sphere1 and sphere3 have " +
                     "different radii");
}  // end if
```

will produce the output

```
sphere1 and sphere3 have different radii
```

which is not true! Both *sphere1* and *sphere3* have a radius of 2.0. Remember that the default *equals* compares two references; they differ here because they reference two distinct objects. If you want to have *equals* check the values contained in the object for equality, you must redefine *equals* in the class. Here is an example of such an *equals* for the class *Sphere*:

Customizing ***equals*** for a class

An ***equals*** method that determines whether two spheres have the same radius

```
public boolean equals(Object rhs) {
   return ((rhs instanceof Sphere) &&
           (theRadius == ((Sphere)rhs).theRadius));
}  // end equals
```

Notice that the parameter of *equals* is of type *Object*. Remember, we are overriding this method as inherited from the class *Object*, and the parameter list and return value must match. Also notice that we are explicitly checking to make sure that the object parameter *rhs* is an instance of the class *Sphere* by using the *instanceof* operator. If the incoming object *rhs* is an instance of the class *Sphere* (or one of its subclasses), *instanceof* will return *true;* otherwise, the operator returns *false*. Thus, the *equals* method will return *false* when *rhs* is of a class other than *Sphere*. If the *instanceof* operator returns *true*, the boolean expression proceeds to check whether the data fields are equal.

In this example, the data field of the class *Sphere* is a primitive type. If an object is used as a data field, *equals* may have to be defined for that object's class as well. It is up to the designer to decide how "deep" the equality checks must be for a particular class.

Java Interfaces

Often it is convenient to be able to specify a set of methods that you might want to provide in many different classes. One way to do this is to define a superclass that contains these methods and then use inheritance to create the different classes that need to provide those methods. This could pose a problem, however, if the subclass also needs to extend another superclass. Java allows only one class to appear in the *extends* clause.

An interface specifies methods and constants but supplies no implementations

To address this situation, Java provides interfaces. An **interface** provides a way to specify methods and constants, but supplies no implementation details for

the methods. Interfaces enable you to specify some desired common behavior that may be useful over many different types of objects. You can then design a method to work with a variety of object types that exhibit this common behavior by specifying the interface as the parameter type for the method, instead of a class. This allows the method to use the common behavior in its implementation, as long as the arguments to the method have implemented the interface.

The Java API has many predefined interfaces. For example, *java.util.Collection* is an interface that provides methods for managing a collection of objects. Here are two of the methods specified in the *Collection* interface:

```
public boolean add(Object o);
public boolean contains(Object o);
```

If you want to have your class provide the methods in this interface, you must indicate your intent to implement the interface by including an ***implements*** clause in your class definition and provide implementations of the methods:

A class that implements an interface

```
public class CardCollection
                        implements java.util.Collection {
  ...
  public boolean add(Object o) {
    // implementation of add method
  } // end add

  public boolean contains(Object o) {
    // implementation of contains method
  }  // end contains

  // and so on...
}  // end CardCollection
```

Suppose there is a *print(Collection c)* method. Instances of *CardCollection* are now eligible to be used as arguments to this method, since *CardCollection* implements the interface *Collection* .

To define your own interface, you use the keyword *interface* instead of *class*, and you provide only method specifications and constants in the interface definition. For example,

An example of an interface

```
public interface MyInterface {
  public final int f1 = 0;
  public void method1();
  public int method2(int a, int b);
}  // end MyInterface
```

This defines an interface *MyInterface* that has one constant *f1* and two methods *method1* and *method2*. Note that the name of the interface ends with

Interface. This is another coding convention that we will use throughout this text.

Interfaces will be used to specify the ADTs that are developed in this text. The implementation of the ADT *list* presented later in this chapter will provide a more complete example of a user-defined interface and how it can be used.

Object Comparison. . Earlier we saw the use of the *equals* method to determine the equality of two objects. Sometimes it is useful to also be able to determine not only the equality of objects, but if one object is greater or less than another object.

When comparing objects in this way, the determination of what makes one object "less" than another object can be specified by implementing the *java.lang.Comparable* interface. This interface contains one method, *compareTo*, that returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object. Here is an example showing the *Sphere* class implementing the *Comparable* interface:

```
class Sphere implements java.lang.Comparable {

  // same methods as before

public int compareTo(Object rhs){
  // Compares rhs object with this object
  // Precondition: The object rhs should be a Sphere object
  // Postcondition: If this sphere has the same radius as the
  // rhs sphere, returns zero. If this sphere has a larger
  // radius than the rhs sphere, a positive integer is
  // returned. If this sphere has a smaller radius than the
  // rhs sphere, a negative integer is returned.
  // Throws: ClassCastException if the rhs object is not a
  // Sphere object.

 // Throws ClassCastException if rhs cannot be cast to Sphere
    Sphere other = (Sphere)rhs;

    if (theRadius == other.theRadius) {
       return 0;  //Equal
    } else if (theRadius < other.theRadius) {
       return -1;
    } else {  // theRadius > other.theRadius
        return 1;
    }
  } // end compareTo
} // end Sphere class
```

In this example, the criterion for comparison is based solely in the radius of the sphere. Spheres with a smaller radius value are considered "less than" spheres with a larger radius. Sometimes, the criterion used to compare objects depends on multiple values. For example, suppose you want to compare the names of people, consisting of a first name and a last name. Simply examining the last name might be sufficient unless you have two people with the same last name, then you would resort to comparing the two first names. The following example defines a *FullName* class, and demonstrates how such a comparison could be defined:

```
public class FullName implements java.lang.Comparable {
  private String firstName;
  private String lastName;

  public FullName(String first, String last) {
    firstName = first;
    lastName = last;
  } // end constructor

  public int compareTo(Object rhs) {
  // Precondition: The object rhs should be a Fullname object
  // Postcondition: Returns 0 if all fields match
  //    if lastName equals rhs.lastName and
  //    firstName is greater than rhs.firstName.
  // Returns -1 if lastName is less than rhs.lastName or
  //    if lastName equals rhs.lastName and
  //    firstName is less than rhs.firstName
  // Throws: ClassCastException if the rhs object is not a Fullname
  // object.

    // Throws ClassCastException if rhs cannot be cast to Fullname
    FullName other = (FullName)rhs;

    if (lastName.compareTo(((FullName)other).lastName)==0){
      return firstName.compareTo(((FullName)other).firstName);
    }
    else {
      return lastName.compareTo(((FullName)other).lastName);
    } // end if
  } // end compareTo
} // end class FullName
```

Java Packages

Java **packages** provide a way to group related classes together. To create a package, you place a *package* statement at the top of each class file that is part of the package. For example, Java source files that are part of a drawing

package would include the following line at the top of the file (Java package names usually begin with a lowercase letter):

```
package drawingPackage;
```

To include a class in a package, begin the class's source file with a ***package*** statement

Just as you must use the same name for both a Java class and the file that contains the class, you must use the same name for a package and the directory that contains all the classes in the package. Thus, the source files for the drawing package must be contained in a directory called *drawingPackage*.

Place the files that contain a package's classes in the same directory

When declaring classes within a package, the keyword *public* must appear in front of the *class* keyword to make the class available to clients of the package. If the drawing package contains a class *Palette*, the source file for *Palette* begins as follows:

```
package drawingPackage;
public class Palette {
...
```

Omitting the keyword *public* will make the class available only to other classes within the package. Sometimes, such restricted access is desirable.

Access to a package's classes can be public or restricted

When a class is publicly available within a package, it can also be used as a superclass for any new class, even those appearing in other packages. This is actually done often within the Java API. For example, the exception class *java.lang.RuntimeException* is the superclass for the class *java.util.NoSuchElementException*.

A package can contain other packages as well. If *shapePackage* is a package in *drawingPackage*, the directory for *shapePackage* must be a subdirectory of the *drawingPackage* directory. The package name consists of the hierarchy of package names, separated by periods. For example, if the Java source file for the class *Sphere* is part of *shapePackage*, the following line must appear at the top of the file *Sphere.java*:

```
package drawingPackage.shapePackage;
```

You already use packages in your programs when you place an *import* statement in your code. When you write a statement such as

```
import java.io.*;
```

Using a package

you indicate to the compiler that you want to use classes in the package *java.io*. The * is a way to indicate that you might use any class in *java.io*, but if you know the specific class from the *java.io* package that you plan to use, you replace the * with the name of that class. For example, the statement

```
import java.io.DataStream;
```

indicates that you will use the class *DataStream* from the package *java.io*.

If you omit the package declaration from the source file for a class, the class is added to a default, unnamed package. If all the classes in a group are declared this way, they are all considered to be within this same unnamed package and hence do not require an *import* statement. But if you are developing a package, and you want to use a class that is contained in the unnamed package, you will need to import the class. In this case, since the package has no name, the class name itself is sufficient in the *import* statement.

An Array-Based Implementation of the ADT List

We will now implement the ADT list as a class. Recall that the ADT list operations are

```
+createList()
+isEmpty():boolean
+size():integer
+add(in index:integer, in newItem:ListItemType)
+remove(in index:integer)
+removeAll()
+get(in index:integer):ListItemType
```

You need to represent the items in the ADT list and its length. Your first thought is probably to store the list's items in an array *items*. In fact, you might believe that the list is simply a fancy name for an array. This belief is not quite true, however. An **array-based implementation** is a natural choice because both an array and a list identify their items by number. However, the ADT list has operations such as *removeAll* that an array does not. In the next chapter you will see another implementation of the ADT list that does not use an array.

In any case, you can store a list's k^{th} item in *items[k-1]*. How much of the array will the list occupy? Possibly all of it, but probably not. That is, you need to keep track of the array elements that you have assigned to the list and those that are available for use in the future. The maximum length of the array—its **physical size**—is a known, fixed value such as *MAX_LIST*. You can keep track of the current number of items in the list—that is, the list's length or **logical size**—in a variable *numItems*. An obvious benefit of this approach is that implementing the operation *size* will be easy. Thus, we could use the following statements to implement a list of integers:

```
private final int MAX_LIST = 100; // max length of list
private int items[MAX_LIST];      // array of list items
private int numItems;             // length of list
```

Shift array elements to insert an item

Figure 4-11 illustrates the data fields for an array-based implementation of an ADT list of integers. To insert a new item at a given position in the array of

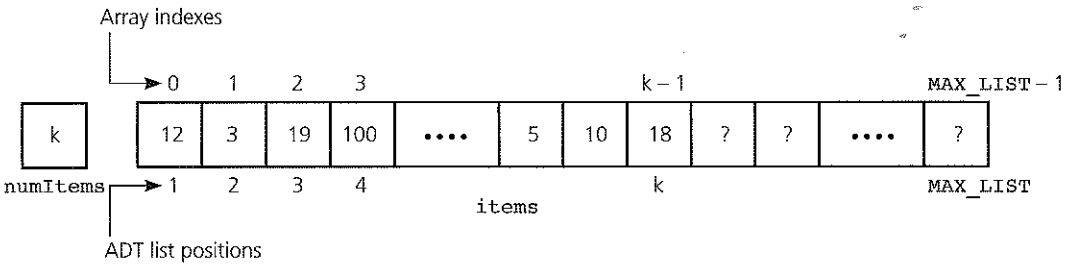

FIGURE 4-11

An array-based implementation of the ADT list

list items, you must shift to the right the items from this position on, and insert the new item in the newly created opening. Figure 4-12 depicts this insertion.

Now consider how to delete an item from the list. You could blank it out, but this strategy can lead to gaps in the array, as Figure 4-13a illustrates. An array that is full of gaps has three significant problems:

- *numItems* – 1 is no longer the index of the last item in the array. You need another variable, *lastPosition*, to contain this index.
- Because the items are spread out, the method *get* might have to look at every cell of the array even when only a few items are present.
- When *items[MAX_LIST - 1]* is occupied, the list could appear full, even when fewer than *MAX_LIST* items are present.

Thus, what you really need to do is shift the elements of the array to fill the gap left by the deleted item, as shown in Figure 4-13b.

Shift array elements to delete an item

You should implement each ADT operation as a method of a class. Each operation will require access to both the array *items* and the list's length *numItems*, so make *items* and *numItems* data fields of the class. To hide *items* and *numItems* from the clients of the class, make these data fields private.

Implement the ADT list as a class

items and ***numItems*** are private data fields

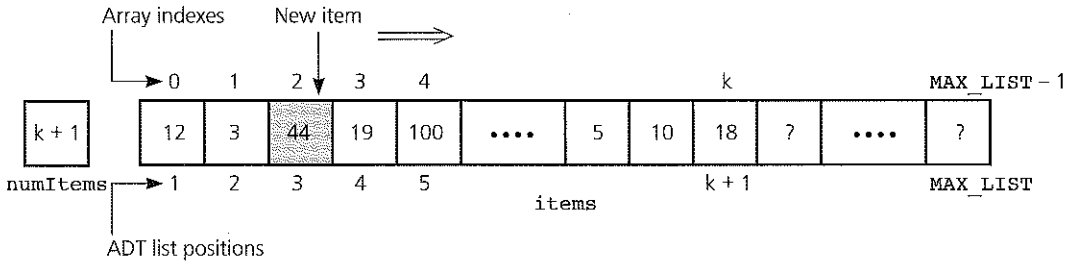

FIGURE 4-12

Shifting items for insertion at position 3

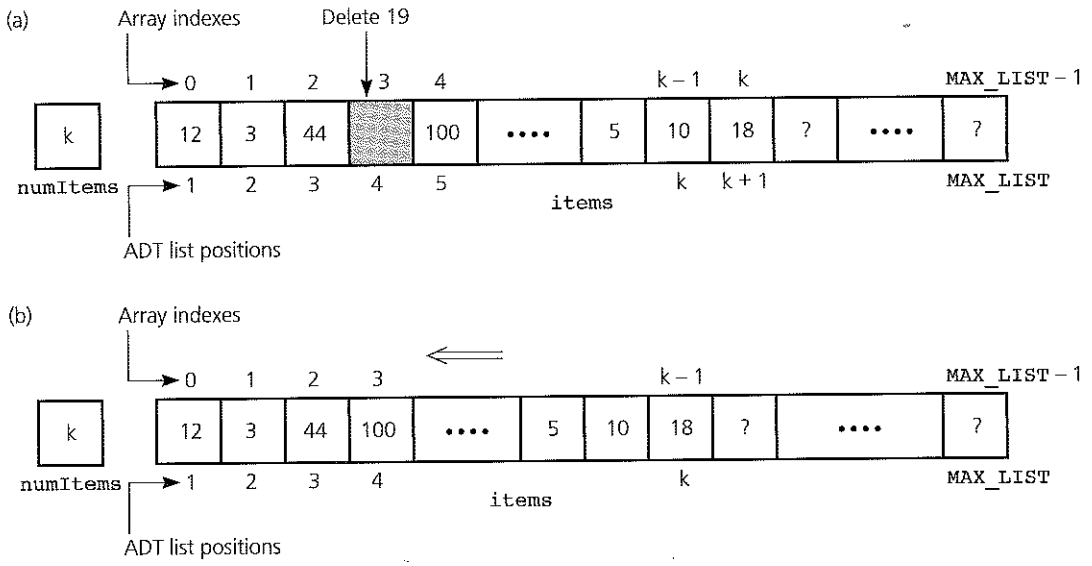

FIGURE 4-13

(a) Deletion causes a gap; (b) fill gap by shifting

translate is a private method

Although it is not obvious why at this point, it will be convenient to define the method *translate(position)*, which returns the index of the array element that contains the list item at position *index*. That is, *translate(position)* returns the index value *position* – 1. Such a method is not one of the ADT operations and should not be available to the client. It simply makes the implementor's task easier. Thus, you should hide *translate* from the client by defining it as a private method.

If one of the operations is provided an index value that is out of range, an exception should be thrown. Here is a definition of an exception that can be used for an out-of-bounds list index called *ListIndexOutOfBoundsException*. It is based upon the more general *IndexOutOfBoundsException* from the Java API:

```
public class ListIndexOutOfBoundsException
             extends IndexOutOfBoundsException {
  public ListIndexOutOfBoundsException(String s) {
    super(s);
  }  // end constructor
}  // end ListIndexOutOfBoundsException
```

Also, the exception *ListException* is needed when the array storing the list becomes full. Here is the exception *ListException*:

```
public class ListException extends RuntimeException {
  public ListException(String s) {
    super(s);
  }  // end constructor
}  // end ListException
```

The following interface *IntegerListInterface* provides the specifications for the list operations. The ADT operation *createList* does not appear in the interface because it will be implemented as a constructor.

Interface for a list of integers

```
// ******************************************************
// Interface IntegerListInterface for the ADT list.
// ******************************************************
public interface IntegerListInterface {
   public boolean isEmpty();
   // Determines whether a list is empty.
   // Precondition: None.
   // Postcondition: Returns true if the list is empty,
   // otherwise returns false.
   // Throws: None.

   public int size();
   // Determines the length of a list.
   // Precondition: None.
   // Postcondition: Returns the number of items that are
   // currently in the list.
   // Throws: None.

   public void removeAll();
   // Deletes all the items from the list.
   // Precondition: None.
   // Postcondition: The list is empty.
   // Throws: None.

   public void add(int index, int item)
            throws ListIndexOutOfBoundsException,
                   ListException;
   // Adds an item to the list at position index.
   // Precondition: index indicates the position at which
   // the item should be inserted in the list.
   // Postcondition: If insertion is successful, item is
   // at position index in the list, and other items are
   // renumbered accordingly.
   // Throws: ListIndexOutOfBoundsException if index < 1 or
   // index > size()+1.
```

```
    // Throws: ListException if item cannot be placed on
    // the list.

    public int get(int index) throws
          ListIndexOutOfBoundsException;
    // Retrieves a list item by position.
    // Precondition: index is the number of the item to be
    // retrieved.
    // Postcondition: If 1 <= index <= size(), the item at
    // position index in the list is returned.
    // Throws: ListIndexOutOfBoundsException if index < 1 or
    // index > size().

    public void remove(int index)
                throws ListIndexOutOfBoundsException;
    // Deletes an item from the list at a given position.
    // Precondition: index indicates where the deletion
    // should occur.
    // Postcondition: If 1 <= index <= size(), the item at
    // position index in the list is deleted, and other items
    // are renumbered accordingly.
    // Throws: ListIndexOutOfBoundsException if index < 1 or
    // index > size().

} // end IntegerListInterface
```

The notion of a list and of the operations that you perform on the list are really independent of the type of items that are stored in the list. The definition just given is very limiting in that it will support only a list of integers. If you use the class `Object` as the type for the list's elements, the specification will be far more flexible. Every class in Java is ultimately derived from the class `Object` through inheritance. This means that any class created in Java could be used as an item in the list class that implements the new list interface.

Use **`Object`** as the type of list elements

What happens if you want to have a list of integers? If the list item type is `Object`, you can no longer use the primitive type `int` as the item type, since `int` is not derived from the class `Object`. In cases where you want a list to contain items of a primitive type, you will need to use a corresponding wrapper class from the Java API. For example, instead of using items of type `int` in the list, the items would be of the type `java.lang.Integer`, a subclass of `Object`.

Here is a revised version, called `ListInterface`, that uses the class `Object` for the list elements. The comments, which are the same as those in `IntegerListInterface`, are left out to save space:

Interface for a list of objects

```
// ******************************************************
// Interface ListInterface for the ADT list.
// ******************************************************
public interface ListInterface {
```

```
  public boolean isEmpty();
  public int size();
  public void add(int index, Object item)
                  throws ListIndexOutOfBoundsException,
                         ListException;
  public Object get(int index)
                    throws ListIndexOutOfBoundsException;
  public void remove(int index)
                     throws ListIndexOutOfBoundsException;
  public void removeAll();
}  // end ListInterface
```

The following class implements the interface *ListInterface*, using arrays.

```
// ********************************************************
// Array-based implementation of the ADT list.
// ********************************************************
public class ListArrayBased implements ListInterface {
```

Implementation file

```
  private static final int MAX_LIST = 50;
  private Object items[];  // an array of list items
  private int numItems;  // number of items in list

  public ListArrayBased() {
    items = new Object[MAX_LIST];
    numItems = 0;
  }  // end default constructor

  public boolean isEmpty() {
    return (numItems == 0);
  } // end isEmpty

  public int size() {
    return numItems;
  }  // end size

  public void removeAll() {
    // Creates a new array; marks old array for
    // garbage collection.
    items = new Object[MAX_LIST];
    numItems = 0;
  } // end removeAll

  public void add(int index, Object item)
                  throws  ListIndexOutOfBoundsException {
    if (numItems > MAX_LIST) {
      throw new ListException("ListException on add");
```

```
    } // end if
    if (index >= 1 && index <= numItems+1) {
      // make room for new element by shifting all items at
      // positions >= index toward the end of the
      // list (no shift if index == numItems+1)
      for (int pos = numItems; pos >= index; pos--) {
          items[translate(pos+1)] = items[translate(pos)];
      } // end for
      // insert new item
      items[translate(index)] = item;
      numItems++;
    }
    else {  // index out of range
      throw new ListIndexOutOfBoundsException(
       "ListIndexOutOfBoundsException on add");
    }  // end if
  } //end add

  public Object get(int index)
                    throws ListIndexOutOfBoundsException {
    if (index >= 1 && index <= numItems) {
      return items[translate(index)];
    }
    else  {  // index out of range
      throw new ListIndexOutOfBoundsException(
        "ListIndexOutOfBoundsException on get");
    }  // end if
  } // end get

  public void remove(int index)
                     throws ListIndexOutOfBoundsException {
    if (index >= 1 && index <= numItems) {
      // delete item by shifting all items at
      // positions > index toward the beginning of the list
      // (no shift if index == size)
      for (int pos = index+1; pos <= size(); pos++) {
        items[translate(pos-1)] = items[translate(pos)];
      }  // end for
      numItems--;
    }
    else {  // index out of range
        throw new ListIndexOutOfBoundsException(
        "ListIndexOutOfBoundsException on remove");
    }  // end if
  } // end remove

  private int translate(int position) {
```

```
    return position - 1;
  }  // end translate
}  // end ListArrayBased
```

The following program segment demonstrates the use of *ListArrayBased*:

```
static public void main(String args[]) {
   . . .
   ListArrayBased aList = new ListArrayBased();
   String dataItem;

   aList.add(1, "Cathryn");
   . . .
   dataItem = aList.get(1);
   . . .
```

Note that references within this program such as *aList.numItems*, *aList.items[4]*, and *aList.translate(6)* would be illegal because *numItems*, *items*, and *translate* are private members of the class.

A client of the class cannot access the class's private members directly

In summary, to implement an ADT, given implementation-independent specifications of the ADT operations, you first must choose a data structure to contain the data. Next, you define and implement a class within a Java source file. The ADT operations are public methods within the class, and the ADT data is represented as data fields that are typically private. You then implement the class's methods within an implementation file. The program that uses the class will be able to access the data only by using the ADT operations.

Summary

1. Data abstraction is a technique for controlling the interaction between a program and its data structures. It builds walls around a program's data structures, just as other aspects of modularity build walls around a program's algorithms. Such walls make programs easier to design, implement, read, and modify.
2. The specification of a set of data-management operations together with the data values upon which they operate define an abstract data type (ADT).
3. The formal mathematical study of ADTs uses systems of axioms to specify the behavior of ADT operations.
4. Only after you have fully defined an ADT should you think about how to implement it. The proper choice of a data structure to implement an ADT depends both on the details of the ADT operations and on the context in which you will use the operations.
5. Even after you have selected a data structure as an implementation for an ADT, the remainder of the program should not depend on your particular choice. That is, you should access the data structure by using only the ADT operations. Thus, you hide the implementation behind a wall of ADT operations. To enforce the wall

within Java, you define the ADT as a class, thus hiding the ADT's implementation from the program that uses the ADT.

6. An object encapsulates both data and operations on that data. In Java, objects are instances of a class, which is a programmer-defined data type.
7. A Java class contains at least one constructor, which is an initialization method.
8. If you do not define a constructor for a class, the compiler will generate a default constructor—that is, one without parameters—for you. Members of a class should be declared as public or private. The client of the class, that is, the program that uses the class, cannot use members that are private. However, the implementations of methods can use them. Typically, you should make the data fields of a class private and provide public methods to access some or all of the data fields.
9. A package provides a mechanism for logically grouping related classes.

Cautions

1. After you design a class, try writing some code that uses your class before you commit to your design. Not only will you see whether your design works for the problem at hand, but you will also test your understanding of your own design and check the comments that document your specifications.
2. When you implement a class, you might discover problems with either your class design or your specifications. If these problems occur, change your design and specifications, try using the class again, and continue implementing. These comments are consistent with the discussion of software life cycle in Chapter 1.
3. A program should not depend upon the particular implementations of its ADTs. By using a class to implement an ADT, you encapsulate the ADT's data and operations. In this way, you can hide implementation details from the program that uses the ADT. In particular, by making the class's data fields private, you can change the class's implementation without affecting the client.
4. By making a class's data fields private, you make it easier to locate errors in a program's logic. An ADT—and hence a class—is responsible for maintaining its data. If an error occurs, you look at the class's implementation for the source of the error. If the client could manipulate this data directly because the data was public, you would not know where to look for errors.
5. Variables that are local to a method's implementation should not be data fields of the class.
6. If you define a constructor for a class but do not also define a default constructor, the compiler will not generate one for you. In this case, a statement such as

```
ListArrayBased aList = new ListArrayBased();
```

   is illegal.
7. An array-based implementation of an ADT restricts the number of items that you can store. Chapter 5 will discuss a way to avoid this problem.

Self-Test Exercises

1. What is the significance of "wall" and "contract"? Why do these notions help you to become a better problem solver?
2. Write a pseudocode method `swap(aList, i, j)` that interchanges the items currently in positions i and j of a list. Define the method in terms of the operations of the ADT list, so that it is independent of any particular implementation of the list. Assume that the list, in fact, has items at positions i and j. What impact does this assumption have on your solution? (See Exercise 2.)
3. What grocery list results from the following sequence of ADT list operations?

```
aList.createList()
aList.add(1, butter)
aList.add(1, eggs)
aList.add(1, milk)
```

4. Write specifications for a list whose insertion, deletion, and retrieval operations are at the end of the list.
5. Write preconditions and postconditions for each of the operations of the ADT sorted list.
6. Write a pseudocode method that creates a sorted list `sortedList` from the list `aList` by using the operations of the ADTs list and sorted list.
7. The specifications of the ADTs list and sorted list do not mention the case in which two or more items have the same value. Are these specifications sufficient to cover this case, or must they be revised?

Exercises

1. Consider an ADT list of integers. Write a method that computes the sum of the integers in the list `aList`. The definition of your method should be independent of the list's implementation.
2. Implement the method `swap`, as described in Self-Test Exercise 2, but remove the assumption that the i^{th} and j^{th} items in the list exist. Throw an exception `ListIndexOutOfBoundsException` if i or j is out of range.
3. Use the method `swap` that you wrote in Exercise 2 to write a method that reverses the order of the items in a list `aList`.
4. The section "The ADT List" describes the methods `displayList` and `replace`. As given in this chapter, these operations exist outside of the ADT; that is, they are not operations of the ADT list. Instead, their implementations are written in terms of the ADT list's operations.
   a. What is an advantage and a disadvantage of the way that `displayList` and `replace` are implemented?
   b. What is an advantage and a disadvantage of adding the operations `displayList` and `replace` to the ADT list?

5. In mathematics, a **set** is a group of distinct items. Specify operations such as equality, subset, union, and intersection as a part of the ADT set.

6. Design and implement an ADT that represents a bank account. The data of the ADT should include the customer name, the account number, and the account balance. The initialization operation should set the data to client-supplied values. Include operations for a deposit and a withdrawal, the addition of interest to the balance, and the display of the statistics of the account.

7. Specify operations that are a part of the ADT fraction. Include typical operations such as addition, subtraction, and reduce (reduce fraction to lowest terms).

8. Specify operations that are a part of the ADT character string. Include typical operations such as length computation and concatenation (appending one string to another).

9. Write pseudocode implementations of the operations of an ADT that represents a rectangle. Include typical operations, such as setting and retrieving the dimensions of the rectangle, finding the area and the perimeter of the rectangle, and displaying the statistics of the rectangle.

10. Implement in Java the pseudocode for the ADT rectangle that you created in Exercise 9.

11. Write a pseudocode method in terms of the ADT appointment book, described in the section "Designing an ADT," for each of the following tasks:

    a. Change the purpose of the appointment at a given date and time.

    b. Display all the appointments for a given date.

    Do you need to add operations to the ADT to perform these tasks?

12. Consider the ADT polynomial—in a single variable x—whose operations include the following:

```
+degree():integer {query}
// Returns the degree of a polynomial.
+getCoefficient(in power:integer):integer
// Returns the coefficient of the x^power term.
+changeCoefficient(in newCoef:integer,
                   in power:integer)
// Replaces the coefficient of the x^power term
// with newCoef.
```

    For this problem, consider only polynomials whose exponents are nonnegative integers. For example,

    $p = 4x^5 + 7x^3 - x^2 + 9$

    The following examples demonstrate the ADT operations on this polynomial.

    `p.degree()` is 5 (the highest power of a term with a nonzero coefficient)

    `p.getCoefficient(3)` is 7 (the coefficient of the x^3 term)

    `p.getCoefficient(4)` is 0 (the coefficient of a missing term is implicitly 0)

    `p.changeCoefficient(-3, 7)` produces the polynomial

    $p = -3x^7 + 4x^5 + 7x^3 - x^2 + 9$

Using these ADT operations, write statements to perform the following tasks:

a. Display the coefficient of the term that has the highest power.

b. Increase the coefficient of the x^3 term by 8.

c. Compute the sum of two polynomials.

13. Write pseudocode implementations of the operations of the ADT polynomial, as defined in Exercise 12, in terms of the operations of the ADT list.

14. Imagine an unknown implementation of an ADT sorted list of integers. This ADT organizes its items into ascending order. Suppose that you have just read N integers into a one-dimensional array of integers called `data`. Write some Java statements that use the operations of the ADT sorted list to sort the array into ascending order.

15. Use the axioms for the ADT list, as given in this chapter in the section "Axioms," to prove that the sequence of operations

```
Insert A into position 2
Insert B into position 2
Insert C into position 2
```

has the same effect on a nonempty list of characters as the sequence

```
Insert C into position 2
Insert B into position 3
Insert A into position 4
```

16. Define a set of axioms for the ADT sorted list and use them to prove that the sorted list of characters, which is defined by the sequence of operations

```
Create an empty sorted list
Insert S
Insert T
Insert R
Delete T
```

is exactly the same as the sorted list defined by the sequence

```
Create an empty sorted list
Insert T
Insert R
Delete T
Insert S
```

17. Repeat Exercise 20 in Chapter 3, using a variation of the ADT list to implement the method $f(n)$.

18. Write pseudocode that merges two sorted lists into a new third sorted list by using only operations of the ADT sorted list.

19. Implement the `List` methods `retrieve` and `remove` to use exceptions.

Programming Problems

1. Design and implement an ADT that represents a triangle. The data for the ADT should include the three sides of the triangle but could also include the triangle's three angles. This data should be declared private in the class that implements the ADT.

   Include at least two initialization operations: One that provides default values for the ADT's data, and another that sets this data to client-supplied values. These operations are the class's constructors.

   The ADT also should include operations that look at the values of the ADT's data; change the values of the ADT's data; compute the triangle's area; and determine whether the triangle is a right triangle, an equilateral triangle, or an isosceles triangle.

2. Design and implement an ADT that represents the time of day. Represent the time as hours and minutes on a 24-hour clock. The hours and minutes are the private data fields of the class that implements the ADT.

   Include at least two initialization operations: One that provides a default value for the time, and another that sets the time to a client-supplied value. These operations are the class's constructors.

   Include operations that set the time, increase the present time by a number of minutes, and display the time in 12-hour and 24-hour notations.

3. Design and implement an ADT that represents a calendar date. You can represent a date's month, day, and year as integers (for example, 4/1/2004). Include operations that advance the date by one day and display the date by using either numbers or words for the months. As an enhancement, include the name of the day.

4. Design and implement an ADT that represents a price in U.S. currency as dollars and cents. After you complete the implementation, write a client method that computes the change due a customer who pays x for an item whose price is y.

5. Define a class for an array-based implementation of the ADT sorted list. Consider a recursive implementation for `locateIndex`. Should `sortedAdd` and `sortedRemove` call `locateIndex`?

6. Write recursive array-based implementations of the insertion, deletion, and retrieval operations for the ADTs list and sorted list.

7. Implement the ADT set that you specified in Exercise 5 by using only arrays and simple variables.

8. Implement the ADT character string that you specified in Exercise 8.

9. Implement the ADT polynomial that Exercise 12 describes.

10. Implement the ADT appointment book, described in the section "Designing an ADT." Add operations as necessary. For example, you should add operations to read and write appointments.

11. a. Implement the ADT fraction that you specified in Exercise 7. Provide operations that read, write, add, subtract, multiply, and divide fractions. The results of all arithmetic operations should be in lowest terms, so include a private method `reduceToLowestTerms`. Exercise 23 in Chapter 2 will help you with the details

of this method. (Should your read and write operations call `reduceToLowestTerms`?) To simplify the determination of a fraction's sign, you can assume that the denominator of the fraction is positive.

b. Specify and implement an ADT for mixed numbers, each of which contains an integer portion and a fractional portion in lowest terms. Assume the existence of the ADT fraction (see part a). Provide operations that read, write, add, subtract, multiply, and divide mixed numbers. The results of all arithmetic operations should have fractional portions that are in lowest terms. Also include an operation that converts a fraction to a mixed number.

12. Implement the recipe database as described in the section "Designing an ADT" and, in doing so, implement the ADTs recipe and measurement. Add operations as necessary. For example, you should add operations to the ADT recipe book to read, write, and scale recipes.

13. Add exception handling to Programming Problem 2 for operations that set or increase the time.

14. Implement a program based on the UML specification in Programming Problem 3 of Chapter 2.

15. Repeat Programming Problem 4 of Chapter 2 in light of your knowledge of ADTs and classes.

16. Repeat Programming Problem 5 of Chapter 2 in light of your knowledge of ADTs and classes.