DATA STRUCTURES & ALGORITHMS

IN JAVA

SECOND EDITION

ROBERT LAFORE

SAMS

Robert Lafore

Data Structures & Algorithms in Java

Second Edition

800 East 96th Street, Indianapolis, Indiana 46240

Data Structures and Algorithms in Java, Second Edition

International Standard Book Number: 0-672-32453-9
Library of Congress Catalog Card Number: 2002106907
Printed in the United States of America
First Printing: December 2002
05 04 03 4 3

Executive Editor *Michael Stephens*
Acquisitions Editor *Carol Ackerman*
Development Editor *Songlin Qiu*
Managing Editor *Charlotte Clapp*
Project Editor *Matt Purcell*
Copy Editor *Chuck Hutchinson*
Indexer *Johnna Dinse*
Proofreader *Cindy Long*
Technical Editor *Mike Kopack*
Team Coordinator *Lynne Williams*
Multimedia Developer *Dan Scherf*
Interior Designer *Gary Adair*
Cover Designer *Alan Clements*
Production *Plan-it Publishing*

Contents at a Glance

Table of Contents

About the Author

Robert Lafore has degrees in Electrical Engineering and Mathematics, has worked as a systems analyst for the Lawrence Berkeley Laboratory, founded his own software company, and is a best-selling writer in the field of computer programming. Some of his current titles are *C++ Interactive Course* and *Object- Oriented Programming in C++*. Earlier best-selling titles include *Assembly Language Primer for the IBM PC and XT* and (back at the beginning of the computer revolution) *Soul of CP/M*.

Dedication

This book is dedicated to my readers, who have rewarded me over the years not only by buying my books, but with helpful suggestions and kind words. Thanks to you all.

Acknowledgments to the First Edition

My gratitude for the following people (and many others) cannot be fully expressed in this short acknowledgment. As always, Mitch Waite had the Java thing figured out before anyone else. He also let me bounce the applets off him until they did the job, and extracted the overall form of the project from a miasma of speculation. My editor, Kurt Stephan, found great reviewers, made sure everyone was on the same page, kept the ball rolling, and gently but firmly ensured that I did what I was supposed to do. Harry Henderson provided a skilled appraisal of the first draft, along with many valuable suggestions. Richard S. Wright, Jr., as technical editor, corrected numerous problems with his keen eye for detail. Jaime Niño, Ph.D., of the University of New Orleans, attempted to save me from myself and occasionally succeeded, but should bear no responsibility for my approach or coding details. Susan Walton has been a staunch and much-appreciated supporter in helping to convey the essence of the project to the non-technical. Carmela Carvajal was invaluable in extending our contacts with the academic world. Dan Scherf not only put the CD-ROM together, but was tireless in keeping me up to date on rapidly evolving software changes. Finally, Cecile Kaufman ably shepherded the book through its transition from the editing to the production process.

Acknowledgments to the Second Edition

My thanks to the following people at Sams Publishing for their competence, effort, and patience in the development of this second edition. Acquisitions Editor Carol Ackerman and Development Editor Songlin Qiu ably guided this edition through the complex production process. Project Editor Matt Purcell corrected a semi-infinite number of grammatical errors and made sure everything made sense. Tech Editor Mike Kopak reviewed the programs and saved me from several problems. Last but not least, Dan Scherf, an old friend from a previous era, provides skilled manage- ment of my code and applets on the Sams Web site.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an executive editor for Sams Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books better.
Please note that I cannot help you with technical problems related to the *topic* of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.
When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments and share them with the author and editors who worked on the book.
Email: feedback@samspublishing.com
Mail: Michael Stephens
Executive Editor Sams Publishing 800 East 96th Street Indianapolis, IN 46240 USA
For more information about this book or another Sams Publishing title, visit our Web site at www.samspublishing.com. Type the ISBN (excluding hyphens) or the title of a book in the Search field to find the page you're looking for.

Introduction

This introduction tells you briefly

• What's new in the Second Edition
• What this book is about
• Why it's different
• Who might want to read it
• What you need to know before you read it
• The software and equipment you need to use it
• How this book is organized

What's New in the Second Edition

This second edition of *Data Structures and Algorithms in Java* has been augmented to make it easier for the reader and for instructors using it as a text in computer science classes. Besides coverage of additional topics, we've added end-of-chapter questions, experiments, and programming projects.

Additional Topics

We've added a variety of interesting new topics to the book. Many provide a basis for programming projects. These new topics include
• Depth-first-search and game simulations
• The Josephus problem
• Huffman codes for data compression
• The Traveling Salesman problem
• Hamiltonian cycles
• The Knight's Tour puzzle
• Floyd's algorithm
• Warshall's algorithm
• 2-3 trees

• The knapsack problem
• Listing N things taken K at a time
• Folding-digits hash functions
• The radix sort

End-of-Chapter Questions

Short questions covering the key points of each chapter are included at the end of each chapter. The answers can be found in Appendix C, "Answers to Questions." These questions are intended as a self-test for readers, to ensure that they have understood the material.

Experiments

We include some suggested activities for the reader. These experiments often involve using the Workshop applets or example programs to examine certain features of an algorithm's operation, but some are pencil-and-paper or "thought experiments."

Programming Projects

Most importantly, we have included at the end of each chapter a number (usually five) of challenging programming projects. They cover a range of difficulty. The easiest are simple variations on the example programs. The most challenging are implementations of topics discussed in the text but for which there are no example programs. Solutions to the Programming Projects are not provided in this book, but see the adjacent note.

NOTE

It is expected that the programming projects will be useful for instructors looking for class assignments. To this end, qualified instructors can obtain suggested solutions to the program- ming projects in the form of source code and executable code. Contact the Sams Web site for information on Instructors Programs.

What This Book Is About

This book is about data structures and algorithms as used in computer programming. Data structures are ways in which data is arranged in your computer's memory (or stored on disk). Algorithms are the procedures a software program uses to manipulate the data in these structures.

Almost every computer program, even a simple one, uses data structures and algo- rithms. For example, consider a program that prints address labels. The program might use an array containing the addresses to be printed and a simple for loop to step through the array, printing each address.

The array in this example is a data structure, and the for loop, used for sequential access to the array, executes a simple algorithm. For uncomplicated programs with small amounts of data, such a simple approach might be all you need. However, for programs that handle even moderately large amounts of data, or which solve prob- lems that are slightly out of the ordinary, more sophisticated techniques are neces- sary. Simply knowing the syntax of a computer language such as Java or C++ isn't enough.

This book is about what you need to know *after* you've learned a programming language. The material we cover here is typically taught in colleges and universities as a second-year course in computer science, after a student has mastered the fundamentals of programming.

What's Different About This Book

There are dozens of books on data structures and algorithms. What's different about this one? Three things:

• Our primary goal in writing this book is to make the topics we cover easy to understand.

• Demonstration programs called *Workshop applets* bring to life the topics we cover, showing you step by step, with "moving pictures," how data structures and algorithms work.

• The example code is written in Java, which is easier to understand than C, C++, or Pascal, the languages traditionally used to demonstrate computer science topics.

Let's look at these features in more detail.

Easy to Understand

Typical computer science textbooks are full of theory, mathematical formulas, and abstruse examples of computer code. This book, on the other hand, concentrates on simple explanations of techniques that can be applied to real-world problems. We avoid complex proofs and heavy math. There are lots of figures to augment the text.

Many books on data structures and algorithms include considerable material on soft- ware engineering. Software engineering is a body of study concerned with designing and implementing large and complex software projects.

However, it's our belief that data structures and algorithms are complicated enough without involving this additional discipline, so we have deliberately de-emphasized software engineering in this book. (We'll discuss the relationship of data structures and algorithms to software engineering in Chapter 1, "Overview.")

Of course, we do use an object-oriented approach, and we discuss various aspects of object-oriented design as we go along, including a mini-tutorial on OOP in Chapter 1. Our primary emphasis, however, is on the data structures and algorithms themselves.

Workshop Applets

From the Sams Web site you can download demonstration programs, in the form of Java applets, that cover the topics we discuss. These applets, which we call *Workshop applets*, will run on most Web browsers. (See Appendix A, "Running the Workshop Applets and Example Programs," for more details.) The Workshop applets create graphic images that show you in "slow motion"

how an algorithm works.
For example, in one Workshop applet, each time you push a button, a bar chart shows you one step in the process of sorting the bars into ascending order. The values of variables used in the sorting algorithm are also shown, so you can see exactly how the computer code works when executing the algorithm. Text displayed in the picture explains what's happening.
Another applet models a binary tree. Arrows move up and down the tree, so you can follow the steps involved in inserting or deleting a node from the tree. There are more than 20 Workshop applets, at least one for each of the major topics in the book.
These Workshop applets make it far more obvious what a data structure really looks like, or what an algorithm is supposed to do, than a text description ever could. Of course, we provide a text description as well. The combination of Workshop applets, clear text, and illustrations should make things easy.
These Workshop applets are standalone graphics-based programs. You can use them as a learning tool that augments the material in the book. Note that they're not the same as the example code found in the text of the book, which we'll discuss next.

NOTE

The Workshop applets, in the form of Java .class files, are available on the Sams Web site at http://www.samspublishing.com/. Enter this book's ISBN (without the hyphens) in the Search box and click Search. When the book's title is displayed, click the title to go to a page where you can download the applets.

Java Example Code

The Java language is easier to understand (and write) than languages such as C and C++. The biggest reason for this is that Java doesn't use pointers. Some people are surprised that pointers aren't necessary for the creation of complex data structures and algorithms. In fact, eliminating pointers makes such code not only easier to write and to understand, but more secure and less prone to errors as well.
Java is a modern object-oriented language, which means we can use an object- oriented approach for the programming examples. This is important, because object- oriented programming (OOP) offers compelling advantages over the old-fashioned procedural approach, and is quickly supplanting it for serious program development. Don't be alarmed if you aren't familiar with OOP. It's not that hard to understand, especially in a pointer-free environment such as Java. We'll explain the basics of OOP in Chapter 1.

NOTE

Like the Workshop applets, the example programs (both source and executable files) can be downloaded from the Sams Web site.

Who This Book Is For

This book can be used as a text in a Data Structures and Algorithms course, typically taught in the second year of a computer science curriculum. However, it is also designed for professional programmers and for anyone else who needs to take the next step up from merely knowing a programming language. Because it's easy to understand, it is also appropriate as a supplemental text to a more formal course.

What You Need to Know Before You Read This Book

The only prerequisite for using this book is a knowledge of some programming language.
Although the example code is written in Java, you don't need to know Java to follow what's happening. Java is not hard to understand, and we've tried to keep the syntax as general as possible, avoiding baroque or Java-specific constructions whenever possible.
Of course, it won't hurt if you're already familiar with Java. Knowing C++ is essen- tially just as good, because Java syntax is based so closely on C++. The differences are minor as they apply to our example programs (except for the welcome elimination of pointers), and we'll discuss them in Chapter 1.

The Software You Need to Use This Book

To run the Workshop applets, you need a Web browser such as Microsoft Internet Explorer or Netscape Communicator. You can also use an applet viewer utility. Applet viewers are available with various Java development systems, including the free system from Sun Microsystems, which we'll discuss in Appendix A.
To run the example programs, you can use the MS-DOS utility in Microsoft Windows (called MS-DOS Prompt) or a similar text-oriented environment.
If you want to modify the source code for the example programs or write your own programs, you'll need a Java development system. Such systems are available commercially, or you can download an excellent basic system from Sun Microsystems, as described in Appendix A.

How This Book Is Organized

This section is intended for teachers and others who want a quick overview of the contents of the book. It assumes you're already familiar with the topics and terms involved in a study of data structures and algorithms.

The first two chapters are intended to ease the reader into data structures and algorithms as painlessly as possible.

Chapter 1, "Overview," presents an overview of the topics to be discussed and intro- duces a small number of terms that will be needed later on. For readers unfamiliar with object-oriented programming, it summarizes those aspects of this discipline that will be needed in the balance of the book, and for programmers who know C++ but not Java, the key differences between these languages are reviewed.

Chapter 2, "Arrays," focuses on arrays. However, there are two subtexts: the use of classes to encapsulate data storage structures and the class interface. Searching, inser- tion, and deletion in arrays and ordered arrays are covered. Linear searching and binary searching are explained. Workshop applets demonstrate these algorithms with unordered and ordered arrays.

In Chapter 3, "Simple Sorting," we introduce three simple (but slow) sorting tech- niques: the bubble sort, selection sort, and insertion sort. Each is demonstrated by a Workshop applet.

Chapter 4, "Stacks and Queues," covers three data structures that can be thought of as Abstract Data Types (ADTs): the stack, queue, and priority queue. These structures reappear later in the book, embedded in various algorithms. Each is demonstrated by a Workshop applet. The concept of ADTs is discussed.

Chapter 5, "Linked Lists," introduces linked lists, including doubly linked lists and double-ended lists. The use of references as "painless pointers" in Java is explained. A Workshop applet shows how insertion, searching, and deletion are carried out.

In Chapter 6, "Recursion," we explore recursion, one of the few chapter topics that is not a data structure. Many examples of recursion are given, including the Towers of Hanoi puzzle and the mergesort, which are demonstrated by Workshop applets.

Chapter 7, "Advanced Sorting," delves into some advanced sorting techniques: Shellsort and quicksort. Workshop applets demonstrate Shellsort, partitioning (the basis of quicksort), and two flavors of quicksort.

In Chapter 8, "Binary Trees," we begin our exploration of trees. This chapter covers the simplest popular tree structure: unbalanced binary search trees. A Workshop applet demonstrates insertion, deletion, and traversal of such trees.

Chapter 9, "Red-Black Trees," explains red-black trees, one of the most efficient balanced trees. The Workshop applet demonstrates the rotations and color switches necessary to balance the tree.

In Chapter 10, "2-3-4 Trees and External Storage," we cover 2-3-4 trees as an example of multiway trees. A Workshop applet shows how they work. We also discuss 2-3 trees and the relationship of 2-3-4 trees to B-trees, which are useful in storing exter- nal (disk) files.

Chapter 11, "Hash Tables," moves into a new field, hash tables. Workshop applets demonstrate several approaches: linear and quadratic probing, double hashing, and separate chaining. The hash-table approach to organizing external files is discussed.

In Chapter 12, "Heaps," we discuss the heap, a specialized tree used as an efficient implementation of a priority queue.

Chapters 13, "Graphs," and 14, "Weighted Graphs," deal with graphs, the first with unweighted graphs and simple searching algorithms, and the second with weighted graphs and more complex algorithms involving the minimum spanning trees and shortest paths.

In Chapter 15, "When to Use What," we summarize the various data structures described in earlier chapters, with special attention to which structure is appropriate in a given situation.

Appendix A, "Running the Workshop Applets and Example Programs," provides details on how to use these two kinds of software. It also tells how to use the Software Development Kit from Sun Microsystems, which can be used to modify the example programs and develop your own programs, and to run the applets and example programs.

Appendix B, "Further Reading," describes some books appropriate for further reading on data structures and other related topics.

Appendix C, "Answers to Questions," contains the answers to the end-of-chapter questions in the text.

Enjoy Yourself!

We hope we've made the learning process as painless as possible. Ideally, it should even be fun. Let us know if you think we've succeeded in reaching this ideal, or if not, where you think improvements might be made.

1

Overview

As you start this book, you may have some questions:

• What are data structures and algorithms?
• What good will it do me to know about them?
• Why can't I just use arrays and for loops to handle my data?
• When does it make sense to apply what I learn here?

This chapter attempts to answer these questions. We'll also introduce some terms you'll need to know and generally set the stage for the more detailed chapters to follow.

Next, for those of you who haven't yet been exposed to an object-oriented language, we'll briefly explain enough about OOP to get you started. Finally, for C++ program- mers who don't know Java we'll point out some of the differences between these languages.

What Are Data Structures and Algorithms Good For?

The subject of this book is data structures and algorithms. A *data structure* is an arrangement of data in a computer's memory (or sometimes on a disk). Data structures include arrays, linked lists, stacks, binary trees, and hash tables, among others. *Algorithms* manipulate the data in these structures in various ways, such as searching for a particu- lar data item and sorting the data.

IN THIS CHAPTER

• What Are Data Structures and Algorithms Good For?
• Overview of Data Structures
• Overview of Algorithms
• Some Definitions
• Object-Oriented Programming
• Software Engineering
• Java for C++ Programmers
• Java Library Data Structures

What sorts of problems can you solve with a knowledge of these topics? As a rough approximation, we might divide the situations in which they're useful into three categories:

• Real-world data storage
• Programmer's tools
• Modeling

These are not hard-and-fast categories, but they may help give you a feeling for the usefulness of this book's subject matter. Let's look at them in more detail.

Real-World Data Storage

Many of the structures and techniques we'll discuss are concerned with how to handle real-world data storage. By real-world data, we mean data that describes phys- ical entities external to the computer. As some examples, a personnel record describes an actual human being, an inventory record describes an existing car part or grocery item, and a financial transaction record describes, say, an actual check written to pay the electric bill.

A non-computer example of real-world data storage is a stack of 3-by-5 index cards. These cards can be used for a variety of purposes. If each card holds a person's name, address, and phone number, the result is an address book. If each card holds the name, location, and value of a household possession, the result is a home inventory.

Of course, index cards are not exactly state-of-the-art. Almost anything that was once done with index cards can now be done with a computer. Suppose you want to update your old index-card system to a computer program. You might find yourself with questions like these:

• How would you store the data in your computer's memory?
• Would your method work for a hundred file cards? A thousand? A million?
• Would your method permit quick insertion of new cards and deletion of old ones?
• Would it allow for fast searching for a specified card?

• Suppose you wanted to arrange the cards in alphabetical order. How would you sort them?

In this book, we will be discussing data structures that might be used in ways similar to a stack of index cards.

Of course, most programs are more complex than index cards. Imagine the database the Department of Motor Vehicles (or whatever it's called in your state) uses to keep track of drivers' licenses, or an airline reservations system that stores passenger and flight information. Such systems may include many data structures. Designing such complex systems requires the application of software engineering techniques, which we'll mention toward the end of this chapter.

Programmer's Tools

Not all data storage structures are used to store real-world data. Typically, real-world data is accessed more or less directly by a program's user. Some data storage struc- tures, however, are not meant to be accessed by the user, but by the program itself. A programmer uses such structures as tools to facilitate some other operation. Stacks, queues, and priority queues are often used in this way. We'll see examples as we go along.

Real-World Modeling

Some data structures directly model real-world situations. The most important data structure of this type is the graph. You can use graphs to represent airline routes between cities or connections in an electric circuit or tasks in a project. We'll cover graphs in Chapter 13, "Graphs," and Chapter 14, "Weighted Graphs." Other data structures, such as stacks and queues, may also be used in simulations. A queue, for example, can model customers waiting in line at a bank or cars waiting at a toll booth.

Overview of Data Structures

Another way to look at data structures is to focus on their strengths and weaknesses. In this section we'll provide an overview, in the form of a table, of the major data storage structures we'll be discussing in this book. This is a bird's-eye view of a land- scape that we'll be covering later at ground level, so don't be alarmed if the terms used are not familiar. Table 1.1 shows the advantages and disadvantages of the various data structures described in this book.

TABLE 1.1 Characteristics of Data Structures

Data Structure	**Advantages**	**Disadvantages**
Array	Quick insertion, very fast access if index known.	Slow search, slow deletion, fixed size.
Ordered array	Quicker search than unsorted array.	Slow insertion and deletion, fixed size.
Stack	Provides last-in, first-out access.	Slow access to other items.
Queue	Provides first-in, first-out access.	Slow access to other items.
Linked list	Quick insertion, quick deletion.	Slow search.
Binary tree	Quick search, insertion, deletion (if tree remains balanced).	Deletion algorithm is complex.
Red-black tree	Quick search, insertion, deletion. Tree always balanced.	Complex.
2-3-4 tree	Quick search, insertion, deletion. Tree always balanced. Similar trees good for disk storage.	Complex.
Hash table	Very fast access if key known. Fast insertion.	Slow deletion, access slow if key not known, inefficient memory usage.
Heap	Fast insertion, deletion, access to largest item.	Slow access to other items.
Graph	Models real-world situations.	Some algorithms are slow and complex.

TABLE 1.1 Continued

The data structures shown in Table 1.1, except the arrays, can be thought of as Abstract Data Types, or ADTs. We'll describe what this means in Chapter 5, "Linked Lists."

Overview of Algorithms

Many of the algorithms we'll discuss apply directly to specific data structures. For most data structures, you need to know how to

• Insert a new data item.

• Search for a specified item.

• Delete a specified item.

You may also need to know how to *iterate* through all the items in a data structure, visiting each one in turn so as to display it or perform some other action on it.

Another important algorithm category is *sorting*. There are many ways to sort data, and we devote Chapter 3, “Simple Sorting,” and Chapter 7, “Advanced Sorting,” to these algorithms.
The concept of *recursion* is important in designing certain algorithms. Recursion involves a method calling itself. We’ll look at recursion in Chapter 6, “Recursion.” (The term *method* is used in Java. In other languages, it is called a function, proce- dure, or subroutine.)

Some Definitions

Let’s look at a few of the terms that we’ll be using throughout this book.

Database

We’ll use the term *database* to refer to all the data that will be dealt with in a particu- lar situation. We’ll assume that each item in a database has a similar format. As an example, if you create an address book using index cards, these cards constitute a database. The term *file* is sometimes used in this sense.

Record *Records* are the units into which a database is divided. They provide a format for storing information. In the index card analogy, each card represents a record. A record includes all the information about some entity, in a situation in which there are many such entities. A record might correspond to a person in a personnel file, a car part in an auto supply inventory, or a recipe in a cookbook file.

Field

A record is usually divided into several *fields*. A field holds a particular kind of data. On an index card for an address book, a person’s name, address, or telephone number is an individual field.
More sophisticated database programs use records with more fields. Figure 1.1 shows such a record, where each line represents a distinct field.
In Java (and other object-oriented languages), records are usually represented by *objects* of an appropriate class. Individual variables within an object represent data fields. Fields within a class object are called *fields* in Java (but *members* in some other languages such as C++).

Employee number: Social security number: Last name: First name: Street address: City: State: Zip code: Phone number: Date of birth: Date of first employment: Salary:
FIGURE 1.1 A record with multiple fields.

Key

To search for a record within a database, you need to designate one of the record’s fields as a *key* (or *search key*). You’ll search for the record with a specific key. For instance, in an address book program, you might search in the name field of each record for the key “Brown.” When you find the record with this key, you can access all its fields, not just the key. We might say that the key *unlocks* the entire record. You could search through the same file using the phone number field or the address field as the key. Any of the fields in Figure 1.1 could be used as a search key.

Object-Oriented Programming

This section is for those of you who haven’t been exposed to object-oriented programming. However, caveat emptor. We cannot, in a few pages, do justice to all the innovative new ideas associated with OOP. Our goal is merely to make it possible for you to understand the example programs in the text.
If, after reading this section and examining some of the example code in the follow- ing chapters, you still find the whole OOP business as alien as quantum physics, you may need a more thorough exposure to OOP. See the reading list in Appendix B, “Further Reading,” for suggestions.

Problems with Procedural Languages

OOP was invented because procedural languages, such as C, Pascal, and early versions of BASIC, were found to be inadequate for large and complex programs. Why was this?

There were two kinds of problems. One was the lack of correspondence between the program and the real world, and the other was the internal organization of the program.
Poor Modeling of the Real World Conceptualizing a real-world problem using procedural languages is difficult. Methods carry out a task, while data stores information, but most real-world objects do both of these things. The thermostat on your furnace, for example, carries out tasks (turning the furnace on and off) but also stores information (the current temperature and

the desired temperature).

If you wrote a thermostat control program in a procedural language, you might end up with two methods, furnace_on() and furnace_off(), but also two global vari- ables, currentTemp (supplied by a thermometer) and desiredTemp (set by the user). However, these methods and variables wouldn't form any sort of programming unit; there would be no unit in the program you could call thermostat. The only such concept would be in the programmer's mind.

For large programs, which might contain hundreds of entities like thermostats, this procedural approach made things chaotic, error-prone, and sometimes impossible to implement at all. What was needed was a better match between things in the program and things in the outside world.

Crude Organizational Units A more subtle, but related, problem had to do with a program's internal organiza- tion. Procedural programs were organized by dividing the code into methods. One difficulty with this kind of method-based organization was that it focused on methods at the expense of data. There weren't many options when it came to data. To simplify slightly, data could be local to a particular method, or it could be global—accessible to all methods. There was no way (at least not a flexible way) to specify that some methods could access a variable and others couldn't.

This inflexibility caused problems when several methods needed to access the same data. To be available to more than one method, such variables needed to be global, but global data could be accessed inadvertently by *any* method in the program. This lead to frequent programming errors. What was needed was a way to fine-tune data accessibility, allowing data to be available to methods with a need to access it, but hiding it from other methods.

Objects in a Nutshell

The idea of *objects* arose in the programming community as a solution to the prob- lems with procedural languages.

Objects Here's the amazing breakthrough that is the key to OOP: An object contains *both methods and variables*. A thermostat object, for example, would contain not only furnace_on() and furnace_off() methods, but also variables called currentTemp and desiredTemp. In Java, an object's variables such as these are called *fields*.

This new entity, the object, solves several problems simultaneously. Not only does an object in a program correspond more closely to an object in the real world, but it also solves the problem engendered by global data in the procedural model. The furnace_on() and furnace_off() methods can access currentTemp and desiredTemp. These variables are hidden from methods that are not part of thermostat, however, so they are less likely to be accidentally changed by a rogue method.

Classes You might think that the idea of an object would be enough for one programming revolution, but there's more. Early on, it was realized that you might want to make several objects of the same type. Maybe you're writing a furnace control program for an entire apartment building, for example, and you need several dozen thermostat objects in your program. It seems a shame to go to the trouble of specifying each one separately. Thus, the idea of classes was born.

A *class* is a specification—a blueprint—for one or more objects. Here's how a thermo- stat class, for example, might look in Java:

```
class thermostat

{private float currentTemp(); private float desiredTemp();
public void furnace_on()

{// method body goes here }
public void furnace_off()

{// method body goes here } } // end class thermostat
```

The Java keyword class introduces the class specification, followed by the name you want to give the class; here it's thermostat. Enclosed in curly brackets are the fields and methods that make up the class. We've left out the bodies of the methods; normally, each would have many lines of program code.

C programmers will recognize this syntax as similar to a structure, while C++ programmers will notice that it's very much like a class in C++, except that there's no semicolon at the end. (Why did we need the semicolon in C++ anyway?)

Creating Objects Specifying a class doesn't create any objects of that class. (In the same way, specify- ing a structure in C doesn't create any variables.) To actually create objects in Java, you must use the keyword new. At the same time an object is created, you need to store a reference to it in a variable of suitable type—that is, the same type as the class.

What's a reference? We'll discuss references in more detail later. In the meantime, think of a reference as a name for an object. (It's actually the object's address, but you don't need to know that.)

Here's how we would create two references to type thermostat, create two new ther- mostat objects, and store references to them in these variables:

```
thermostat therm1, therm2; // create two references
therm1 = new thermostat(); // create two objects and therm2 = new thermostat(); // store references to them
```

Incidentally, creating an object is also called *instantiating* it, and an object is often referred to as an *instance* of a class.

Accessing Object Methods After you specify a class and create some objects of that class, other parts of your program need to interact with these objects. How do they do that?

Typically, other parts of the program interact with an object's methods, not with its data (fields). For example, to tell the therm2 object to turn on the furnace, we would say

```
therm2.furnace_on();
```

The dot operator (.) associates an object with one of its methods (or occasionally with one of its fields).

At this point we've covered (rather telegraphically) several of the most important features of OOP. To summarize:

• Objects contain both methods and fields (data).

• A class is a specification for any number of objects.

• To create an object, you use the keyword new in conjunction with the class name.

• To invoke a method for a particular object, you use the dot operator.

These concepts are deep and far reaching. It's almost impossible to assimilate them the first time you see them, so don't worry if you feel a bit confused. As you see more classes and what they do, the mist should start to clear.

A Runnable Object-Oriented Program

Let's look at an object-oriented program that runs and generates actual output. It features a class called BankAccount that models a checking account at a bank. The program creates an account with an opening balance, displays the balance, makes a deposit and a withdrawal, and then displays the new balance. Listing 1.1 shows bank.java.

LISTING 1.1 The bank.java Program

```
// bank.java // demonstrates basic OOP syntax // to run this program: C>java BankApp //////////////////////////////////////////////////////////////// class
BankAccount

{private double balance; // account balance
public BankAccount(double openingBalance) // constructor

{balance = openingBalance; }
public void deposit(double amount) // makes deposit

{balance = balance + amount; }
public void withdraw(double amount) // makes withdrawal

{balance = balance - amount; }
public void display() // displays balance

{System.out.println("balance=" + balance); } } // end class BankAccount ////////////////////////////////////////////////////////////////
```

LISTING 1.1 Continued

```
class BankApp

{public static void main(String[] args)

{BankAccount ba1 = new BankAccount(100.00); // create acct
System.out.print("Before transactions, "); ba1.display(); // display balance
ba1.deposit(74.35); // make deposit ba1.withdraw(20.00); // make withdrawal
System.out.print("After transactions, "); ba1.display(); // display balance } // end main() } // end class BankApp
```

Here's the output from this program:

```
Before transactions, balance=100 After transactions, balance=154.35
```

There are two classes in bank.java. The first one, BankAccount, contains the fields and methods for our bank account. We'll examine it in detail in a moment. The second class, BankApp, plays a special role.

The BankApp **Class** To execute the program in Listing 1.1 from an MS-DOS prompt, you type java BankApp following the C: prompt:

```
C:\>java BankApp
```

This command tells the java interpreter to look in the BankApp class for the method called main(). Every Java application must have a main() method; execution of the program starts at the beginning of main(), as you can see in Listing 1.1. (You don't need to worry yet about the String[] args argument in main().)

The main() method creates an object of class BankAccount, initialized to a value of 100.00, which is the opening balance, with this statement:

```
BankAccount ba1 = new BankAccount(100.00); // create acct
```

The System.out.print() method displays the string used as its argument, Before transactions:, and the account displays its balance with this statement:

```
ba1.display();
```

The program then makes a deposit to, and a withdrawal from, the account:

```
ba1.deposit(74.35); ba1.withdraw(20.00);
```

Finally, the program displays the new account balance and terminates.

The BankAccount **Class** The only data field in the BankAccount class is the amount of money in the account, called balance. There are three methods. The deposit() method adds an amount to the balance, withdrawal() subtracts an amount, and display() displays the balance.

Constructors The BankAccount class also features a *constructor*, which is a special method that's called automatically whenever a new object is created. A constructor always has exactly the same name as the class, so this one is called BankAccount(). This constructor has one argument, which is used to set the opening balance when the account is created.

A constructor allows a new object to be initialized in a convenient way. Without the constructor in this program, you would have needed an additional call to deposit() to put the opening balance in the account.

Public and Private Notice the keywords public and private in the BankAccount class. These keywords are *access modifiers* and determine which methods can access a method or field. The balance field is preceded by private. A field or method that is private can be accessed only by methods that are part of the same class. Thus, balance cannot be accessed by statements in main() because main() is not a method in BankAccount.

All the methods in BankAccount have the access modifier public, however, so they can be accessed by methods in other classes. That's why statements in main() can call deposit(), withdrawal(), and display().

Data fields in a class are typically made private and methods are made public. This protects the data; it can't be accidentally modified by methods of other classes. Any outside entity that needs to access data in a class must do so using a method of the same class. Data is like a queen bee, kept hidden in the middle of the hive, fed and cared for by worker-bee methods.

Inheritance and Polymorphism

We'll briefly mention two other key features of object-oriented programming: inheri- tance and polymorphism.

Inheritance is the creation of one class, called the *extended* or derived class, from another class called the *base* class. The extended class has all the features of the base class, plus some additional features. For example, a secretary class might be derived from a more general employee class and include a field called typingSpeed that the employee class lacked.

In Java, inheritance is also called *subclassing*. The base class may be called the *super- class*, and the extended class may be called the *subclass*.

Inheritance enables you to easily add features to an existing class and is an impor- tant aid in the design of programs with many related classes. Inheritance thus makes it easy to reuse classes for a slightly different purpose, a key benefit of OOP.

Polymorphism involves treating objects of different classes in the same way. For poly- morphism to work, these different classes must be derived from the same base class. In practice, polymorphism usually involves a method call that actually executes different methods for objects of different classes.

For example, a call to display() for a secretary object would invoke a display method in the secretary class, while the exact same call for a manager object would invoke a different display method in the manager class. Polymorphism simplifies and clarifies program design and coding.

For those not familiar with them, inheritance and polymorphism involve significant additional complexity. To keep the focus on data structures and algorithms, we have avoided these features in our example programs. Inheritance and polymorphism are important and powerful aspects of OOP but are not necessary for the explanation of data structures and algorithms.

Software Engineering

In recent years, it has become fashionable to begin a book on data structures and algorithms with a chapter on software engineering. We don't follow that approach, but let's briefly examine software engineering and see how it fits into the topics we discuss in this book.

Software engineering is the study of ways to create large and complex computer programs, involving many programmers. It focuses on the overall design of the programs and on the creation of that design from the needs of the end users. Software engineering is concerned with the life cycle of a software project, which includes specification, design, verification, coding, testing, production, and maintenance.

It's not clear that mixing software engineering on one hand and data structures and algorithms on the other actually helps the student understand either topic. Software engineering is rather abstract and is difficult to grasp until you've been involved yourself in a large project. The use of data structures and algorithms, on the other hand, is a nuts-and-bolts discipline concerned with the details of coding and data storage.

Accordingly, we focus on the essentials of data structures and algorithms. How do they really work? What structure or algorithm is best in a particular situation? What do they look like translated into Java code? As we noted, our intent is to make the material as easy to understand as possible. For further reading, we mention some books on software engineering in Appendix B.

Java for C++ Programmers

If you're a C++ programmer who has not yet encountered Java, you might want to read this section. We'll mention several ways that Java differs from C++.

This section is not intended to be a primer on Java. We don't even cover all the differences between C++ and Java. We're interested in only a few Java features that might make it hard for C++ programmers to figure out what's going on in the example programs.

No Pointers

The biggest difference between C++ and Java is that Java doesn't use pointers. To a C++ programmer, not using pointers may at first seem quite amazing. How can you get along without pointers?

Throughout this book we'll use pointer-free code to build complex data structures. You'll see that this approach is not only possible, but actually easier than using C++ pointers.

Actually, Java only does away with *explicit* pointers. Pointers, in the form of memory addresses, are still there, under the surface. It's sometimes said that, in Java, *every- thing* is a pointer. This statement is not completely true, but it's close. Let's look at the details.

References Java treats primitive data types (such as int, float, and double) differently than objects. Look at these two statements:

```
int intVar; // an int variable called intVar BankAccount bc1; // reference to a BankAccount object
```

In the first statement, a memory location called intVar actually holds a numerical value such as 127 (assuming such a value has been placed there). However, the memory location bc1 does not hold the data of a BankAccount object. Instead, it contains the *address* of a BankAccount object that is actually stored elsewhere in memory. The name bc1 is a *reference to* this object; it's not the object itself.

Actually, bc1 won't hold a reference if it has not been assigned an object at some prior point in the program. Before being assigned an object, it holds a reference to a special object called null. In the same way, intVar won't hold a numerical value if it's never been assigned one. The compiler will complain if you try to use a variable that has never been assigned a value.

In C++, the statement

```
BankAccount bc1;
```

actually creates an object; it sets aside enough memory to hold all the object's data. In Java, all this statement creates is a place to put an object's memory address. You can think of a reference as a pointer with the syntax of an ordinary variable. (C++ has reference variables, but they must be explicitly specified with the & symbol.)

Assignment It follows that the assignment operator (=) operates differently with Java objects than with C++ objects. In C++, the statement

```
bc2 = bc1;
```

copies all the data from an object called bc1 into a different object called bc2. Following this statement, there are two objects with the same data. In Java, on the other hand, this same assignment statement copies the memory address that bc1 refers to into bc2.

Both bc1 and bc2 now refer to exactly the same object; they are references to it.

This can get you into trouble if you're not clear what the assignment operator does. Following the assignment statement shown above, the statement

```
bc1.withdraw(21.00);
```

and the statement

```
bc2.withdraw(21.00);
```

both withdraw $21 from *the same bank account object*.

Suppose you actually want to copy data from one object to another. In this case you must make sure you have two separate objects to begin with and then copy each field separately. The equal sign won't do the job.

The new **Operator** Any object in Java must be created using new. However, in Java, new returns a refer- ence, not a pointer as in C++. Thus, pointers aren't necessary to use new. Here's one way to create an object:

```
BankAccount ba1; ba1 = new BankAccount();
```

Eliminating pointers makes for a more secure system. As a programmer, you can't find out the actual address of ba1, so you can't accidentally corrupt it. However, you probably don't need to know it, unless you're planning something wicked.

How do you release memory that you've acquired from the system with new and no longer need? In C++, you use delete. In Java, you don't need to worry about releas- ing memory. Java periodically looks through each block of memory that was obtained with new to see if valid references to it still exist. If there are no such refer- ences, the block is returned to the free memory store. This process is called *garbage collection*.

In C++ almost every programmer at one time or another forgets to delete memory blocks, causing "memory leaks" that consume system resources, leading to bad performance and even crashing the system. Memory leaks can't happen in Java (or at least hardly ever).

Arguments In C++, pointers are often used to pass objects to functions to avoid the overhead of copying a large object. In Java, objects are always passed as references. This approach also avoids copying the object:

```
void method1()

{BankAccount ba1 = new BankAccount(350.00); method2(ba1); }
void method2(BankAccount acct)

{}
```

In this code, the references ba1 and acct both refer to the same object. In C++ acct would be a separate object, copied from ba1.

Primitive data types, on the other hand, are always passed by value. That is, a new variable is created in the method and the value of the argument is copied into it.

Equality and Identity In Java, if you're talking about primitive types, the equality operator (==) will tell you whether two variables have the same value:

```
int intVar1 = 27; int intVar2 = intVar1; if(intVar1 == intVar2)
System.out.println("They're equal");
```

This is the same as the syntax in C and C++, but in Java, because relational operators use references, they work differently with objects. The equality operator, when applied to objects, tells you whether two references are identical—that is, whether they refer to the same object:

```
carPart cp1 = new carPart("fender"); carPart cp2 = cp1; if(cp1 == cp2)
System.out.println("They're Identical");
```

In C++ this operator would tell you if two objects contained the same data. If you want to see whether two objects contain the same data in Java, you must use the equals() method of the Object class:

```
carPart cp1 = new carPart("fender"); carPart cp2 = cp1; if( cp1.equals(cp2) )
System.out.println("They're equal");
```

This technique works because all objects in Java are implicitly derived from the Object class.

Overloaded Operators

This point is easy: There are no overloaded operators in Java. In C++, you can rede- fine +, *, =, and most other operators so that they behave differently for objects of a particular class. No such redefinition is possible in Java. Use a named method instead, such as add() or whatever.

Primitive Variable Types

The primitive or built-in variable types in Java are shown in Table 1.2.

TABLE 1.2 Primitive Data Types

Name	**Size in Bits**	**Range of Values**
boolean	1	true or false
byte	8	–128 to +127
char	16	'\u0000' to '\uFFFF'
short	16	–32,768 to +32,767
int	32	–2,147,483,648 to +2,147,483,647
long	64	–9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
float	32	Approximately 10^{-38} to 10^{+38}; 7 significant digits
double	64	Approximately 10^{-308} to 10^{+308}; 15 significant digits

Unlike C and C++, which use integers for true/false values, boolean is a distinct type in Java.

Type char is unsigned, and uses two bytes to accommodate the Unicode character representation scheme, which can handle international characters.

The int type varies in size in C and C++, depending on the specific computer plat- form; in Java an int is always 32 bits.

Literals of type float use the suffix F (for example, 3.14159F); literals of type double need no suffix. Literals of type long use suffix L (as in 45L); literals of the other integer types need no suffix.

Java is more strongly typed than C and C++; many conversions that were automatic in those languages require an explicit cast in Java.

All types not shown in Table 1.2, such as String, are classes.

Input/Output

There have been changes to input/output as Java has evolved. For the console-mode applications we'll be using as example programs in this book, some clunky-looking but effective constructions are available for input and output. They're quite different from the workhorse cout and cin approaches in C++ and printf() and scanf() in C.

Older versions of the Java Software Development Kit (SDK) required the line

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

at the beginning of the source file for all input/output routines. Now this line is needed only for input.

Output You can send any primitive type (numbers and characters), and String objects as well, to the display with these statements:

```
System.out.print(var); // displays var, no linefeed System.out.println(var); // displays var, then starts new line
```

The print() method leaves the cursor on the same line; println() moves it to the beginning of the next line.

In older versions of the SDK, a System.out.print() statement did not actually write anything to the screen. It had to be followed by a System.out.println()or System.out.flush() statement to display the entire buffer. Now it displays immediately.

You can use several variables, separated by plus signs, in the argument. Suppose in this statement the value of ans is 33:

```
System.out.println("The answer is " + ans);
```

Then the output will be

```
The answer is 33
```

Inputting a String Input is considerably more involved than output. In general, you want to read any input as a String object. If you're actually inputting something else, say a character or number, you then convert the String object to the desired type.

As we noted, any program that uses input must include the statement

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

at the beginning of the program. Without this statement, the compiler will not recognize such entities as IOException and InputStreamReader.

String input is fairly baroque. Here's a method that returns a string entered by the user:

```
public static String getString() throws IOException

{InputStreamReader isr = new InputStreamReader(System.in); BufferedReader br = new BufferedReader(isr); String s = br.readLine(); return s; }
```

This method returns a String object, which is composed of characters typed on the keyboard and terminated with the Enter key. The details of the InputStreamReader and BufferedReader classes need not concern us here.

Besides importing java.io.*, you'll need to add throws IOException to all input methods, as shown in the preceding code. In fact, you'll need to add throws IOException to any method, such as main(), that calls any of the input methods.

Inputting a Character Suppose you want your program's user to enter a character. (By *enter*, we mean typing something and pressing the Enter key.) The user may enter a single character or (incorrectly) more than one. Therefore, the safest way to read a character involves reading a String and picking off its first character with the charAt() method:

```
public static char getChar() throws IOException
{String s = getString(); return s.charAt(0); }
```

The charAt() method of the String class returns a character at the specified position in the String object; here we get the first character, which is number 0. This approach prevents extraneous characters being left in the input buffer. Such charac- ters can cause problems with subsequent input.

Inputting Integers To read numbers, you make a String object as shown before and convert it to the type you want using a conversion method. Here's a method, getInt(), that converts input into type int and returns it:

```
public int getInt() throws IOException
{String s = getString(); return Integer.parseInt(s); }
```

The parseInt() method of class Integer converts the string to type int. A similar routine, parseLong(), can be used to convert type long. In older versions of the SDK, you needed to use the line

```
import java.lang.Integer;
```

at the beginning of any program that used parseInt(), but this convention is no longer necessary.

For simplicity, we don't show any error-checking in the input routines in the example programs. The user must type appropriate input, or an exception will occur. With the code shown here the exception will cause the program to terminate. In a serious program you should analyze the input string before attempting to convert it and should also catch any exceptions and process them appropriately.

Inputting Floating-Point Numbers Types float and double can be handled in somewhat the same way as integers, but the conversion process is more complex. Here's how you read a number of type double:

```
public int getDouble() throws IOException
{String s = getString(); Double aDub = Double.valueOf(s); return aDub.doubleValue(); }
```

The String is first converted to an object of type Double (uppercase *D*), which is a "wrapper" class for type double. A method of Double called doubleValue() then converts the object to type double.

For type float, there's an equivalent Float class, which has equivalent valueOf() and floatValue() methods.

Java Library Data Structures

The java.util package contains data structures, such as Vector (an extensible array), Stack, Dictionary, and Hashtable. In this book we'll usually ignore these built-in classes. We're interested in teaching fundamentals, not the details of a particular implementation. However, occasionally we'll find some of these structures useful. You must use the line

```
import java.util.*;
```

before you can use objects of these classes.

Although we don't focus on them, such class libraries, whether those that come with Java or others available from third-party developers, can offer a rich source of versa- tile, debugged storage classes. This book should equip you with the knowledge to know what sort of data structure you need and the fundamentals of how it works. Then you can decide whether you should write your own classes or use someone else's.

Summary

• A data structure is the organization of data in a computer's memory or in a disk file.
• The correct choice of data structure allows major improvements in program efficiency.
• Examples of data structures are arrays, stacks, and linked lists.
• An algorithm is a procedure for carrying out a particular task.
• In Java, an algorithm is usually implemented by a class method.
• Many of the data structures and algorithms described in this book are most often used to build databases.
• Some data structures are used as programmer's tools: They help execute an algorithm.
• Other data structures model real-world situations, such as telephone lines running between cities.
• A database is a unit of data storage composed of many similar records.
• A record often represents a real-world object, such as an employee or a car part.
• A record is divided into fields. Each field stores one characteristic of the object described by the record.

• A key is a field in a record that's used to carry out some operation on the data. For example, personnel records might be sorted by a LastName field.
• A database can be searched for all records whose key field has a certain value. This value is called a search key.

Questions

These questions are intended as a self-test for readers. Answers to the questions may be found in Appendix C.

1. In many data structures you can ________ a single record, _________ it, and _______ it.

2. Rearranging the contents of a data structure into a certain order is called _________ .

3. In a database, a field is
a. a specific data item.
b. a specific object.
c. part of a record.
d. part of an algorithm.

4. The field used when searching for a particular record is the ______________ .

5. In object-oriented programming, an object
a. is a class.
b. may contain data and methods.
c. is a program.
d. may contain classes.

6. A class
a. is a blueprint for many objects.
b. represents a specific real-world object.
c. will hold specific values in its fields.
d. specifies the type of a method.

7. In Java, a class specification
a. creates objects.
b. requires the keyword new.
c. creates references.
d. none of the above.

8. When an object wants to do something, it uses a ________ .

9. In Java, accessing an object's methods requires the _____ operator.

10. In Java, boolean and byte are _____________ .

(There are no experiments or programming projects for Chapter 1.)

2

Arrays

The array is the most commonly used data storage struc- ture; it's built into most programming languages. Because arrays are so well known, they offer a convenient jumping- off place for introducing data structures and for seeing how object-oriented programming and data structures relate to one another. In this chapter we'll introduce arrays in Java and demonstrate a home-made array class.

We'll also examine a special kind of array, the ordered array, in which the data is stored in ascending (or descend- ing) key order. This arrangement makes possible a fast way of searching for a data item: the binary search.

We'll start the chapter with a Java Workshop applet that shows insertion, searching, and deletion in an array. Then we'll show some sample Java code that carries out these same operations.

Later we'll examine ordered arrays, again starting with a Workshop applet. This applet will demonstrate a binary search. At the end of the chapter we'll talk about Big O notation, the most widely used measure of algorithm efficiency.

The Array Workshop Applet

Suppose you're coaching kids-league baseball, and you want to keep track of which players are present at the prac- tice field. What you need is an attendance-monitoring program for your laptop—a program that maintains a data- base of the players who have shown up for practice. You can use a simple data structure to hold this data. There are several actions you would like to be able to perform:

• Insert a player into the data structure when the player arrives at the field.

IN THIS CHAPTER

• The Basics of Arrays in Java
• Dividing a Program into Classes
• Class Interfaces
• Java Code for an Ordered Array
• Logarithms
• Storing Objects
• Big O Notation
• Why Not Use Arrays for Everything?

• Check to see whether a particular player is present, by searching for the player's number in the structure.

• Delete a player from the data structure when that player goes home.

These three operations—insertion, searching, and deletion—will be the fundamental ones in most of the data storage structures we'll study in this book.

We'll often begin the discussion of a particular data structure by demonstrating it with a Workshop applet. This approach will give you a feeling for what the structure and its algorithms do, before we launch into a detailed explanation and demonstrate sample code. The Workshop applet called Array shows how an array can be used to implement insertion, searching, and deletion.

Now start up the Array Workshop applet, as described in Appendix A, "Running the Workshop Applets and Example Programs," with

```
C:\>appletviewer Array.html
```

Figure 2.1 shows the resulting array with 20 elements, 10 of which have data items in them. You can think of these items as representing your baseball players. Imagine that each player has been issued a team shirt with the player's number on the back. To make things visually interesting, the shirts come in a variety of colors. You can see each player's number and shirt color in the array.

FIGURE 2.1 The Array Workshop applet.

This applet demonstrates the three fundamental procedures mentioned earlier:

• The Ins button inserts a new data item.
• The Find button searches for specified data item.
• The Del button deletes a specified data item.

Using the New button, you can create a new array of a size you specify. You can fill this array with as many data items as you want using the Fill button. Fill creates a set of items and randomly assigns them numbers and colors. The numbers are in the range 0 to 999. You can't create an array of more than 60 cells, and you can't, of course, fill more data items than there are array cells.

Also, when you create a new array, you'll need to decide whether duplicate items will be allowed; we'll return to this question in a moment. The default value is no dupli- cates, so the No Dups radio button is initially selected to indicate this setting.

Insertion

Start with the default arrangement of 20 cells and 10 data items, and the No Dups button selected. You insert a baseball player's number into the array when the player arrives at the practice field, having been dropped off by a parent. To insert a new item, press the Ins button once. You'll be prompted to enter the value of the item:

```
Enter key of item to insert
```

Type a number, say 678, into the text field in the upper-right corner of the applet. (Yes, it is hard to get three digits on the back of a kid's shirt.) Press Ins again and the applet will confirm your choice:

Will insert item with key 678

A final press of the button will cause a data item, consisting of this value and a random color, to appear in the first empty cell in the array. The prompt will say something like

Inserted item with key 678 at index 10

Each button press in a Workshop applet corresponds to a step that an algorithm carries out. The more steps required, the longer the algorithm takes. In the Array Workshop applet the insertion process is very fast, requiring only a single step. This is true because a new item is always inserted in the first vacant cell in the array, and the algorithm knows this location because it knows how many items are already in the array. The new item is simply inserted in the next available space. Searching and deletion, however, are not so fast.

In no-duplicates mode you're on your honor not to insert an item with the same key as an existing item. If you do, the applet displays an error message, but it won't prevent the insertion. The assumption is that you won't make this mistake.

Searching

To begin a search, click the Find button. You'll be prompted for the key number of the person you're looking for. Pick a number that appears on an item somewhere in the middle of the array. Type in the number and repeatedly press the Find button. At each button press, one step in the algorithm is carried out. You'll see the red arrow start at cell 0 and move methodically down the cells, examining a new one each time you press the button. The index number in the message

Checking next cell, index = 2

will change as you go along. When you reach the specified item, you'll see the message

Have found item with key 505

or whatever key value you typed in. Assuming duplicates are not allowed, the search will terminate as soon as an item with the specified key value is found.

If you have selected a key number that is not in the array, the applet will examine every occupied cell in the array before telling you that it can't find that item.

Notice that (again assuming duplicates are not allowed) the search algorithm must look through an average of half the data items to find a specified item. Items close to the beginning of the array will be found sooner, and those toward the end will be found later. If N is the number of items, the average number of steps needed to find an item is N/2. In the worst-case scenario, the specified item is in the last occupied cell, and N steps will be required to find it.

As we noted, the time an algorithm takes to execute is proportional to the number of steps, so searching takes much longer on the average (N/2 steps) than insertion (one step).

Deletion

To delete an item, you must first find it. After you type in the number of the item to be deleted, repeated button presses will cause the arrow to move, step by step, down the array until the item is located. The next button press deletes the item, and the cell becomes empty. (Strictly speaking, this step isn't necessary because we're going to copy over this cell anyway, but deleting the item makes it clearer what's happening.)

Implicit in the deletion algorithm is the assumption that *holes* are not allowed in the array. A hole is one or more empty cells that have filled cells above them (at higher index numbers). If holes are allowed, all the algorithms become more complicated because they must check to see whether a cell is empty before examining its contents. Also, the algorithms become less efficient because they must waste time looking at unoccupied cells. For these reasons, occupied cells must be arranged contiguously: no holes allowed.

Therefore, after locating the specified item and deleting it, the applet must shift the contents of each subsequent cell down one space to fill in the hole. Figure 2.2 shows an example.

Item to be deleted

0 1 2 3 4 5 6 7 8 9

84 61 15 73 26 38 11 49 53 32

❶ ❷ ❹❸ 0 1 2 3 4 5 6 7 8

84 61 15 73 26 11 49 53 32

Contents shifted down

FIGURE 2.2 Deleting an item.

If the item in cell 5 (38, in Figure 2.2) is deleted, the item in 6 shifts into 5, the item in 7 shifts into 6, and so on to the last

occupied cell. During the deletion process, when the item is located, the applet shifts down the contents of the higher-indexed cells as you continue to press the Del button.
A deletion requires (assuming no duplicates are allowed) searching through an average of N/2 elements and then moving the remaining elements (an average of N/2 moves) to fill up the resulting hole. This is N steps in all.

The Duplicates Issue

When you design a data storage structure, you need to decide whether items with duplicate keys will be allowed. If you're working with a personnel file and the key is an employee number, duplicates don't make much sense; there's no point in assigning the same number to two employees. On the other hand, if the key value is last names, then there's a distinct possibility several employees will have the same key value, so duplicates should be allowed.

Of course, for the baseball players, duplicate numbers should not be allowed. Keeping track of the players would be hard if more than one wore the same number.
The Array Workshop applet lets you select either option. When you use New to create a new array, you're prompted to specify both its size and whether duplicates are permitted. Use the radio buttons Dups OK or No Dups to make this selection.
If you're writing a data storage program in which duplicates are not allowed, you may need to guard against human error during an insertion by checking all the data items in the array to ensure that none of them already has the same key value as the item being inserted. This check is inefficient, however, and increases the number of steps required for an insertion from one to N. For this reason, our applet does not perform this check.

Searching with Duplicates Allowing duplicates complicates the search algorithm, as we noted. Even if it finds a match, it must continue looking for possible additional matches until the last occu- pied cell. At least, this is one approach; you could also stop after the first match. How you proceed depends on whether the question is "Find me everyone with blue eyes" or "Find me someone with blue eyes."
When the Dups OK button is selected, the applet takes the first approach, finding all items matching the search key. This approach always requires N steps because the algorithm must go all the way to the last occupied cell.

Insertion with Duplicates Insertion is the same with duplicates allowed as when they're not: A single step inserts the new item. But remember, if duplicates are not allowed, and there's a possibility the user will attempt to input the same key twice, you may need to check every existing item before doing an insertion.

Deletion with Duplicates Deletion may be more complicated when duplicates are allowed, depending on exactly how "deletion" is defined. If it means to delete only the first item with a specified value, then, on the average, only N/2 comparisons and N/2 moves are necessary. This is the same as when no duplicates are allowed.
If, however, deletion means to delete *every* item with a specified key value, the same operation may require multiple deletions. Such an operation will require checking N cells and (probably) moving more than N/2 cells. The average depends on how the duplicates are distributed throughout the array.
The applet assumes this second meaning and deletes multiple items with the same key. This is complicated because each time an item is deleted, subsequent items must be shifted farther. For example, if three items are deleted, then items beyond the last

deletion will need to be shifted three spaces. To see how this operation works, set the applet to Dups OK and insert three or four items with the same key. Then try delet- ing them.
Table 2.1 shows the average number of comparisons and moves for the three opera- tions, first where no duplicates are allowed and then where they are allowed. N is the number of items in the array. Inserting a new item counts as one move.

TABLE 2.1 Duplicates OK Versus No Duplicates

	No Duplicates	**Duplicates OK**
Search	N/2 comparisons	N comparisons
Insertion	No comparisons, one move	No comparisons, one move
Deletion	N/2 comparisons, N/2 moves	N comparisons, more than N/2 moves

You can explore these possibilities with the Array Workshop applet.
The difference between N and N/2 is not usually considered very significant, except when you're fine-tuning a program. Of more importance, as we'll discuss toward the end of this chapter, is whether an operation takes one step, N steps, log(N) steps, or N^2 steps.

Not Too Swift

One of the significant things to notice when you're using the Array applet is the slow and methodical nature of the algorithms. With the exception of insertion, the algorithms involve stepping through some or all of the cells in the array. Different data structures offer much faster (but more complex) algorithms. We'll see one, the binary search on an ordered array, later in this chapter, and others throughout this book.

The Basics of Arrays in Java

The preceding section showed graphically the primary algorithms used for arrays. Now we'll see how to write programs to carry out these algorithms, but we first want to cover a few of the fundamentals of arrays in Java.

If you're a Java expert, you can skip ahead to the next section, but even C and C++ programmers should stick around. Arrays in Java use syntax similar to that in C and C++ (and not that different from other languages), but there are nevertheless some unique aspects to the Java approach.

Creating an Array

As we noted in Chapter 1, "Overview," there are two kinds of data in Java: primitive types (such as int and double) and objects. In many programming languages (even object-oriented ones such as C++), arrays are primitive types, but in Java they're treated as objects. Accordingly, you must use the new operator to create an array:

```
int[] intArray; // defines a reference to an array intArray = new int[100]; // creates the array, and
// sets intArray to refer to it
```

Or you can use the equivalent single-statement approach:

```
int[] intArray = new int[100];
```

The [] operator is the sign to the compiler we're naming an array object and not an ordinary variable. You can also use an alternative syntax for this operator, placing it after the name instead of the type:

```
int intArray[] = new int[100]; // alternative syntax
```

However, placing the [] after the int makes it clear that the [] is part of the type, not the name.

Because an array is an object, its name—intArray in the preceding code—is a refer- ence to an array; it's not the array itself. The array is stored at an address elsewhere in memory, and intArray holds only this address.

Arrays have a length field, which you can use to find the size (the number of elements) of an array:

```
int arrayLength = intArray.length; // find array size
```

As in most programming languages, you can't change the size of an array after it's been created.

Accessing Array Elements

Array elements are accessed using an index number in square brackets. This is similar to how other languages work:

```
temp = intArray[3]; // get contents of fourth element of array intArray[7] = 66; // insert 66 into the eighth cell
```

Remember that in Java, as in C and C++, the first element is numbered 0, so that the indices in an array of 10 elements run from 0 to 9.

If you use an index that's less than 0 or greater than the size of the array less 1, you'll get the Array Index Out of Bounds runtime error.

Initialization

Unless you specify otherwise, an array of integers is automatically initialized to 0 when it's created. Unlike C++, this is true even of arrays defined within a method (function). Say you create an array of objects like this:

```
autoData[] carArray = new autoData[4000];
```

Until the array elements are given explicit values, they contain the special null object. If you attempt to access an array element that contains null, you'll get the runtime error Null Pointer Assignment. The moral is to make sure you assign something to an element before attempting to access it.

You can initialize an array of a primitive type to something besides 0 using this syntax:

```
int[] intArray = { 0, 3, 6, 9, 12, 15, 18, 21, 24, 27 };
```

Perhaps surprisingly, this single statement takes the place of both the reference decla- ration and the use of new to create the array. The numbers within the curly brackets are called the *initialization list*. The size of the array is determined by the number of values in this list.

An Array Example

Let's look at some example programs that show how an array can be used. We'll start with an old-fashioned procedural version and then show the equivalent object- oriented approach. Listing 2.1 shows the old-fashioned version, called array.java.

LISTING 2.1 The array.java Program

```
// array.java // demonstrates Java arrays // to run this program: C>java arrayApp //////////////////////////////////////////////////////////////// class ArrayApp
```

```
{public static void main(String[] args)

{long[] arr; // reference to array arr = new long[100]; // make array int nElems = 0; // number of items
```

LISTING 2.1 Continued

```
int j; // loop counter long searchKey; // key of item to search for //--------------------------------------------------------------
arr[0] = 77; // insert 10 items arr[1] = 99; arr[2] = 44; arr[3] = 55; arr[4] = 22; arr[5] = 88; arr[6] = 11; arr[7] = 00; arr[8] = 66; arr[9] = 33;
nElems = 10; // now 10 items in array //--------------------------------------------------------------
for(j=0; j<nElems; j++) // display items
System.out.print(arr[j] + " "); System.out.println(""); //--------------------------------------------------------------
searchKey = 66; // find item with key 66 for(j=0; j<nElems; j++) // for each element,
if(arr[j] == searchKey) // found item?
break; // yes, exit before end if(j == nElems) // at the end?
System.out.println("Can't find " + searchKey); // yes elseSystem.out.println("Found " + searchKey); // no
//--------------------------------------------------------------
searchKey = 55; // delete item with key 55 for(j=0; j<nElems; j++) // look for it if(arr[j] == searchKey)
break; for(int k=j; k<nElems-1; k++) // move higher ones down
arr[k] = arr[k+1]; nElems--; // decrement size //--------------------------------------------------------------
for(j=0; j<nElems; j++) // display items
System.out.print( arr[j] + " "); System.out.println(""); } // end main() } // end class ArrayApp
```

In this program, we create an array called arr, place 10 data items (kids' numbers) in it, search for the item with value 66 (the shortstop, Louisa), display all the items, remove the item with value 55 (Freddy, who had a dentist appointment), and then display the remaining 9 items. The output of the program looks like this:

```
77 99 44 55 22 88 11 0 66 33 Found 66 77 99 44 22 88 11 0 66 33
```

The data we're storing in this array is type long. We use long to make it clearer that this is data; type int is used for index values. We've chosen a primitive type to simplify the coding. Generally, the items stored in a data structure consist of several fields, so they are represented by objects rather than primitive types. We'll see such an example toward the end of this chapter.

Insertion Inserting an item into the array is easy; we use the normal array syntax:

```
arr[0] = 77;
```

We also keep track of how many items we've inserted into the array with the nElems variable.

Searching The searchKey variable holds the value we're looking for. To search for an item, we step through the array, comparing searchKey with each element. If the loop variable j reaches the last occupied cell with no match being found, the value isn't in the array. Appropriate messages are displayed: Found 66 or Can't find 27.

Deletion Deletion begins with a search for the specified item. For simplicity, we assume (perhaps rashly) that the item is present. When we find it, we move all the items with higher index values down one element to fill in the "hole" left by the deleted element, and we decrement nElems. In a real program, we would also take appropri- ate action if the item to be deleted could not be found.

Display Displaying all the elements is straightforward: We step through the array, accessing each one with arr[j] and displaying it.

Program Organization The organization of array.java leaves something to be desired. The program has only one class, ArrayApp, and this class has only one method, main(). array.java is essentially an old-fashioned procedural program. Let's see if we can make it easier to understand (among other benefits) by making it more object oriented.

We're going to provide a gradual introduction to an object-oriented approach, using two steps. In the first, we'll separate the data storage structure (the array) from the rest of the program. The remaining part of the program will become a *user* of the structure. In the second step, we'll improve the communication between the storage structure and its user.

Dividing a Program into Classes

The array.java program in Listing 2.1 essentially consists of one big method. We can reap many benefits by dividing the program into classes. What classes? The data storage structure itself is one candidate, and the part of the program that uses this data structure is another. By dividing the program into these two classes, we can clarify the functionality of the program, making it easier to design and understand (and in real programs to modify and maintain).

In array.java we used an array as a data storage structure, but we treated it simply as a language element. Now we'll encapsulate the array in a class, called LowArray. We'll also provide class methods by which objects of other classes (the LowArrayApp class in this case) can access the array. These methods allow communication between LowArray and LowArrayApp.
Our first design of the LowArray class won't be entirely successful, but it will demon- strate the need for a better approach. The lowArray.java program in Listing 2.2 shows how it looks.

LISTING 2.2 The lowArray.java Program

```
// lowArray.java // demonstrates array class with low-level interface // to run this program: C>java LowArrayApp
//////////////////////////////////////////////////////////////// class LowArray
{private long[] a; // ref to array a //--------------------------------------------------------------
public LowArray(int size) // constructor
{ a = new long[size]; } // create array //-------------------------------------------------------------- public void setElem(int index, long value) // set value
{ a[index] = value; } //--------------------------------------------------------------
public long getElem(int index) // get value
{ return a[index]; } //--------------------------------------------------------------
```

LISTING 2.2 Continued

```
} // end class LowArray //////////////////////////////////////////////////////////////// class LowArrayApp
{public static void main(String[] args)
{LowArray arr; // reference arr = new LowArray(100); // create LowArray object int nElems = 0; // number of items in array int j; // loop variable
arr.setElem(0, 77); // insert 10 items arr.setElem(1, 99); arr.setElem(2, 44); arr.setElem(3, 55); arr.setElem(4, 22); arr.setElem(5, 88);
arr.setElem(6, 11); arr.setElem(7, 00); arr.setElem(8, 66); arr.setElem(9, 33); nElems = 10; // now 10 items in array
for(j=0; j<nElems; j++) // display items System.out.print(arr.getElem(j) + " "); System.out.println("");
int searchKey = 26; // search for data item for(j=0; j<nElems; j++) // for each element,
if(arr.getElem(j) == searchKey) // found item?
break; if(j == nElems) // no
System.out.println("Can't find " + searchKey); else // yes
System.out.println("Found " + searchKey);
// delete value 55 for(j=0; j<nElems; j++) // look for it if(arr.getElem(j) == 55)
break; for(int k=j; k<nElems; k++) // higher ones down
```

LISTING 2.2 Continued

```
arr.setElem(k, arr.getElem(k+1) ); nElems--; // decrement size
for(j=0; j<nElems; j++) // display items
System.out.print( arr.getElem(j) + " "); System.out.println(""); } // end main() } // end class LowArrayApp
////////////////////////////////////////////////////////////////
```

The output from the lowArray.java program is similar to that from array.java, except that we try to find a non-existent key value (26) before deleting the item with the key value 55:

```
77 99 44 55 22 88 11 0 66 33 Can't find 26 77 99 44 22 88 11 0 66 33
```

Classes LowArray and LowArrayApp

In lowArray.java, we essentially wrap the class LowArray around an ordinary Java array. The array is hidden from the outside world inside the class; it's private, so only LowArray class methods can access it. There are three LowArray methods: setElem() and getElem(), which insert and retrieve an element, respectively; and a constructor, which creates an empty array of a specified size.
Another class, LowArrayApp, creates an object of the LowArray class and uses it to store and manipulate data. Think of LowArray as a tool and LowArrayApp as a user of the tool. We've divided the program into two classes with clearly defined roles. This is a valuable first step in making a program object oriented.
A class used to store data objects, as is LowArray in the lowArray.java program, is sometimes called a *container class*. Typically, a container class not only stores the data but also provides methods for accessing the data and perhaps also sorting it and performing other complex actions on it.

Class Interfaces

We've seen how a program can be divided into separate classes. How do these classes interact with each other? Communication between classes and the division of responsibility between them are important aspects of object-oriented programming.

Private Data
Interface
FIGURE 2.3 The LowArray interface.

Not So Convenient

The interface to the LowArray class in lowArray.java is not particularly convenient. The methods setElem() and getElem() operate on a low conceptual level, perform- ing exactly the same tasks as the [] operator in an ordinary Java array. The class user, represented by the main() method in the LowArrayApp class, ends up having to carry out the same low-level operations it did in the non-class version of an array in the

This point is especially true when a class may have many different users. Typically, a class can be used over and over by different users (or the same user) for different purposes. For example, someone might use the LowArray class in some other program to store the serial numbers of his traveler's checks. The class can handle this task just as well as it can store the numbers of baseball players.

If a class is used by many different programmers, the class should be designed so that it's easy to use. The way that a class user relates to the class is called the class *inter- face*. Because class fields are typically private, when we talk about the interface, we usually mean the class methods—what they do and what their arguments are. By calling these methods, a class user interacts with an object of the class. One of the important advantages conferred by object-oriented programming is that a class inter- face can be designed to be as convenient and efficient as possible. Figure 2.3 is a fanciful interpretation of the LowArray interface.

a
setElem()
getElem()
lowArray

array.java program. The only difference was that it related to setElem() and getElem() instead of the [] operator. It's not clear that this approach is an improvement.

Also notice that there's no convenient way to display the contents of the array. Somewhat crudely, the LowArrayApp class simply uses a for loop and the getElem() method for this purpose. We could avoid repeated code by writing a separate method for LowArrayApp that it could call to display the array contents, but is it really the responsibility of the LowArrayApp class to provide this method?

Thus, lowArray.java demonstrates how to divide a program into classes, but it really doesn't buy us too much in practical terms. Let's see how to redistribute responsibili- ties between the classes to obtain more of the advantages of OOP.

Who's Responsible for What?

In the lowArray.java program, the main()routine in the LowArrayApp class, the user of the data storage structure, must keep track of the indices to the array. For some users of an array, who need random access to array elements and don't mind keeping track of the index numbers, this arrangement might make sense. For example, sorting an array, as we'll see in the next chapter, can make efficient use of this direct hands-on approach.

In a typical program, however, the user of the data storage device won't find access to the array indices to be helpful or relevant.

The highArray.java Example

Out next example program shows an improved interface for the storage structure class, called HighArray. Using this interface, the class user (the HighArrayApp class) no longer needs to think about index numbers. The setElem() and getElem() methods are gone; they're replaced by insert(), find(), and delete(). These new methods don't require an index number as an argument because the class takes responsibility for handling index numbers. The user of the class (HighArrayApp) is free to concentrate on the *what* instead of the *how*—what's going to be inserted, deleted, and accessed, instead of exactly how these activities are carried out.

Figure 2.4 shows the HighArray interface, and Listing 2.3 shows the highArray.java program.

FIGURE 2.4 The HighArray interface.

LISTING 2.3 The highArray.java Program

```
// highArray.java // demonstrates array class with high-level interface // to run this program: C>java HighArrayApp
//////////////////////////////////////////////////////////////// class HighArray
{private long[] a; // ref to array a private int nElems; // number of data items //----------------------------------------------------------- public
```

```
HighArray(int max) // constructor
{a = new long[max]; // create the array nElems = 0; // no items yet } //----------------------------------------------------------- public boolean find(long
searchKey)
{ // find specified value int j; for(j=0; j<nElems; j++) // for each element,
if(a[j] == searchKey) // found item?
```

Private Data
a

Find()
Interface
insert() delete()

highArray()

LISTING 2.3 Continued

```
break; // exit loop before end if(j == nElems) // gone to end?
return false; // yes, can't find it elsereturn true; // no, found it } // end find() //----------------------------------------------------------- public void
insert(long value) // put element into array
{a[nElems] = value; // insert it nElems++; // increment size } //----------------------------------------------------------- public boolean delete(long value)
{int j; for(j=0; j<nElems; j++) // look for it
if( value == a[j] )
break; if(j==nElems) // can't find it
return false; else // found it
{for(int k=j; k<nElems; k++) // move higher ones down
a[k] = a[k+1]; nElems--; // decrement size return true; } } // end delete() //----------------------------------------------------------- public void display()
// displays array contents {for(int j=0; j<nElems; j++) // for each element,
System.out.print(a[j] + " "); // display it System.out.println(""); } //----------------------------------------------------------- } // end class HighArray
//////////////////////////////////////////////////////////////// class HighArrayApp
{public static void main(String[] args)
```

LISTING 2.3 Continued

```
{int maxSize = 100; // array size HighArray arr; // reference to array arr = new HighArray(maxSize); // create the array
arr.insert(77); // insert 10 items arr.insert(99); arr.insert(44); arr.insert(55); arr.insert(22); arr.insert(88); arr.insert(11); arr.insert(00);
arr.insert(66); arr.insert(33);
arr.display(); // display items
int searchKey = 35; // search for item if( arr.find(searchKey) )
                    System.out.println("Found " + searchKey); elseSystem.out.println("Can't find " + searchKey);
arr.delete(00); // delete 3 items arr.delete(55); arr.delete(99);
arr.display(); // display items again } // end main() } // end class HighArrayApp ////////////////////////////////////////////////////////////////
```

The HighArray class is now wrapped around the array. In main(), we create an array of this class and carry out almost the same operations as in the lowArray.java program: We insert 10 items, search for an item—one that isn't there—and display the array contents. Because deleting is so easy, we delete 3 items (0, 55, and 99) instead of 1 and finally display the contents again. Here's the output:

```
77 99 44 55 22 88 11 0 66 33 Can't find 35 77 44 22 88 11 66 33
```

Notice how short and simple main() is. The details that had to be handled by main() in lowArray.java are now handled by HighArray class methods.
In the HighArray class, the find() method looks through the array for the item whose key value was passed to it as an argument. It returns true or false, depending on whether it finds the item.

The insert() method places a new data item in the next available space in the array. A field called nElems keeps track of the number of array cells that are actually filled with data items. The main() method no longer needs to worry about how many items are in the array.

The delete() method searches for the element whose key value was passed to it as an argument and, when it finds that element, shifts all the elements in higher index cells down one cell, thus writing over the deleted value; it then decrements nElems.

We've also included a display() method, which displays all the values stored in the array.

The User's Life Made Easier

In lowArray.java (Listing 2.2), the code in main() to search for an item required eight lines; in highArray.java, it requires only one. The class user, the HighArrayApp class, need not worry about index numbers or any other array details. Amazingly, the class user doesn't even need to know *what kind of data structure* the HighArray class is using to store the data. The structure is hidden behind the interface. In fact, in the next section, we'll see the same interface used with a somewhat different data structure.

Abstraction

The process of separating the *how* from the *what*—how an operation is performed inside a class, as opposed to what's visible to the class user—is called *abstraction*. Abstraction is an important aspect of software engineering. By abstracting class functionality, we make it easier to design a program because we don't need to think about implementation details at too early a stage in the design process.

The Ordered Workshop Applet

Imagine an array in which the data items are arranged in order of ascending key values—that is, with the smallest value at index 0, and each cell holding a value larger than the cell below. Such an array is called an *ordered array*.

When we insert an item into this array, the correct location must be found for the insertion: just above a smaller value and just below a larger one. Then all the larger values must be moved up to make room.

Why would we want to arrange data in order? One advantage is that we can speed up search times dramatically using a *binary search*.

Start the Ordered Workshop applet, using the procedure described in Chapter 1. You'll see an array; it's similar to the one in the Array Workshop applet, but the data is ordered. Figure 2.5 shows this applet.

FIGURE 2.5 The Ordered Workshop applet.

In the ordered array we've chosen not to allow duplicates. As we saw earlier, this decision speeds up searching somewhat but slows down insertion.

Linear Search

Two search algorithms are available for the Ordered Workshop applet: linear and binary. Linear search is the default. Linear searches operate in much the same way as the searches in the unordered array in the Array applet: The red arrow steps along, looking for a match. The difference is that in the ordered array, the search quits if an item with a larger key is found.

Try out a linear search. Make sure the Linear radio button is selected. Then use the Find button to search for a non-existent value that, if it were present, would fit somewhere in the middle of the array. In Figure 2.5, this number might be 400. You'll see that the search terminates when the first item larger than 400 is reached; it's 427 in the figure. The algorithm knows there's no point looking further.

Try out the Ins and Del buttons as well. Use Ins to insert an item with a key value that will go somewhere in the middle of the existing items. You'll see that insertion requires moving all the items with key values larger than the item being inserted.

Use the Del button to delete an item from the middle of the array. Deletion works much the same as it did in the Array applet, shifting items with higher index numbers down to fill in the hole left by the deletion. In the ordered array, however, the deletion algorithm can quit partway through if it doesn't find the item, just as the search routine can.

Binary Search

The payoff for using an ordered array comes when we use a binary search. This kind of search is much faster than a linear search, especially for large arrays.

The Guess-a-Number Game Binary search uses the same approach you did as a kid (if you were smart) to guess a number in the well-known children's guessing game. In this game, a friend asks you to guess a number she's thinking of between 1 and 100. When you guess a number, she'll tell you one of three things: Your guess is larger than the number she's think- ing of, it's smaller, or you guessed correctly.

To find the number in the fewest guesses, you should always start by guessing 50. If your friend says your guess is too low, you deduce the number is between 51 and 100, so your next guess should be 75 (halfway between 51 and 100). If she says it's too high, you deduce the number is between 1 and 49, so your next guess should be 25.

Each guess allows you to divide the range of possible values in half. Finally, the range is only one number long, and that's the answer.

Notice how few guesses are required to find the number. If you used a linear search, guessing first 1, then 2, then 3, and so on, finding the number would take you, on the average, 50 guesses. In a binary search each guess divides the range of possible values in half, so the number of guesses required is far fewer. Table 2.2 shows a game session when the number to be guessed is 33.

TABLE 2.2 Guessing a Number

Step Number	Number Guessed	Result	Range of Possible Values
0			1–100
1	50	Too high	1–49
2	25	Too low	26–49
3	37	Too high	26–36
4	31	Too low	32–36
5	34	Too high	32–33
6	32	Too low	33–33
7	33	Correct	

The correct number is identified in only seven guesses. This is the maximum. You might get lucky and guess the number before you've worked your way all the way down to a range of one. This would happen if the number to be guessed was 50, for example, or 34.

Binary Search in the Ordered Workshop Applet To perform a binary search with the Ordered Workshop applet, you must use the New button to create a new array. After the first press, you'll be asked to specify the size of the array (maximum 60) and which kind of searching scheme you want: linear or binary. Choose binary by clicking the Binary radio button. After the array is created, use the Fill button to fill it with data items. When prompted, type the amount (not more than the size of the array). A few more presses fills in all the items.

When the array is filled, pick one of the values in the array and see how you can use the Find button to locate it. After a few preliminary presses, you'll see the red arrow pointing to the algorithm's current guess, and you'll see the range shown by a verti-cal blue line adjacent to the appropriate cells. Figure 2.6 depicts the situation when the range is the entire array.

FIGURE 2.6 Initial range in the binary search.

At each press of the Find button, the range is halved and a new guess is chosen in the middle of the range. Figure 2.7 shows the next step in the process.