Great Ideas in Computer Science with JAVA

Alan W. **Biermann** and Dietolf **Ramm**

Alan W. Biermann and Dietolf Ramm

Great Ideas in Computer Science with Java

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Times New Roman by Asco Typesetters, Hong Kong, and printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Biermann, Alan W., 1939–
Great ideas in computer science with Java / Alan W. Biermann and Dietolf Ramm.
p. cm.
Includes index.
ISBN 0-262-02497-7 (pbk. : alk. paper)
1. Java (Computer program language) I. Ramm, Dietolf. II. Title.
QA76.73.J38 B52 2001
005.2'762—dc21 2001030635

Contents

book can also serve as a text in the only computer science course that some students will ever take. It provides a conceptual structure of computing and information technology that well-informed lay people should have. It supports the model of FITness (Fluency in Information Technology) described in a recent National Research Council study (Snyder et al. 1999) by covering most of the information technology concepts that the study specified for current-day fluency.

A Thousand Heroes

This book is the product of fifteen years of experience in teaching "great ideas in computer science" at Duke University and many other institutions. The list of contributors includes many faculty and student assistants who have taught the "great ideas" approach. (See, for example, a description by Biermann of this type of course at several universities, in "Computer Science for the Many," *Computer* 27 (1994): 62–73.) Our teaching assistants have contributed extensively by helping us develop an approach to introducing Java, by writing many of the notes that eventually evolved into this book, and by developing the laboratory exercises and software for our classes. The primary contributors were Steve Myers, Eric Jewart, Steve Ruby, and Greg Keim. We owe special thanks to our faculty colleagues Owen Astrachan, Robert Duvall, Jeff Forbes, and Gershon Kedem for providing constructive critique, stimulating conversation, and technical advice. Ben Allen prepared some of the Java programs that are presented in the simulation chapter. Carrie Liken created some of the graphics in chapter 5. Matt Evans was the artist for most of the cartoons. Charlene Gordon contributed cartoons for chapters five and eleven. Other contributors have been David and Jennifer Biermann, Alice M. Gordon, Karl, Lenore, and M. K. Ramm, Jeifu Shi, Michael Fulkerson, Elina Kaplan, Denita Thomas, our several thousand students in courses at Duke, long lists of people who helped us with our earlier editions, our manuscript editors Deborah Cantor-Adams and Alice Cheyer, and, as always, our kind executive editor Robert Prior.

Studying Academic Computer Science: An Introduction

Rumors

Computers are the subject of many rumors, and we wonder what to believe. People say that computers in the future will do all clerical jobs and even replace some well-trained experts. They say computers are beginning to simulate the human mind, to create art, to prove theorems, to learn, and to make careful judgments. They say that computers will permeate every aspect of our jobs and private lives by managing communication, manipulating information, and providing entertainment. They say that even our political systems will be altered—that in previously closed societies computers will bring universal communication that will threaten the existing order, and in free societies they will bring increased monitoring and control. On the other hand, there are skeptics who say that computer science has many limitations and that the impact of machines has been overstated.

Some of these rumors are correct and give us fair warning of things to come. Others may be somewhat fanciful, leading us to worry about the future more than is necessary. Still others point out questions that we may argue about for years without finding answers. Whatever the case, we can be sure that there are many important issues related to computers that are of vital importance, and they are worth trying to understand.

We should study computer science and address these concerns. We should get our hands on a machine and try to make it go. We should control the machine; we should play with it; we should harness it; and most important, we should try to understand how it works. We should try to build insights from our limited experiences that will illuminate answers to our questions. We should try to arm ourselves with understanding because the Computer Age is upon us.

This book is designed to help people understand computers and computer science. It begins with a study of programming in the belief that using, controlling, and manipulating machines is an essential avenue to understanding them. Then it takes readers on a guided tour of the internals of a machine, exploring all of its essential functioning from the internal

registers and memory to the software that controls them. Finally, the book explores the limitations of computing, the frontiers of the science as they are currently understood.

In short, the book attempts to give a thorough introduction to the field with an emphasis on the fundamental mechanisms that enable computers to work. It presents many of the "great ideas" of computer science, the intellectual paradigms that scientists use to understand the field. These ideas provide the tools to help readers comprehend and live with machines.

Studying Computer Science

Computer science is the study of recipes and ways to carry them out. A recipe is a procedure or method for doing something. The science studies kinds of recipes, the properties of recipes, languages for writing them down, methods for creating them, and the construction of machines that will carry them out. Of course, computer scientists want to distinguish themselves from chefs, so they have their own name for recipes: they call them *algorithms*. But we will save most of the technical jargon for later.

If we wish to understand computer science, then we must study recipes, or algorithms. The first problem relates to how to conceive of them and how to write them down. For example, one might want a recipe for treating a disease, for classifying birds on the basis of their characteristics, or for organizing a financial savings program. We need to study some example recipes to see how they are constructed, and then we need to practice writing our own. We need experience in abstracting the essence of real-world situations and in organizing this knowledge into a sequence of steps for getting our tasks done.

Once we have devised a method for doing something, we wish to *code* it in a computer language in order to communicate our desires to the machine. Thus, it is necessary to learn a computer language and to learn to translate the steps of a recipe into commands that can be carried out by a machine. This book presents a language called *Java*, which is easy to learn and quite satisfactory for our example programs.

The combination of creating the recipe and coding it into a computer language is called *programming*, and this is the subject of the first part of the book (chapters 1–6). These chapters give a variety of examples of problem types, their associated solution methods, and the Java code, the *program*, required to solve them. Chapter 7 discusses problems related to scaling up the lessons learned here to industrial-sized programming projects.

While the completion of the programming chapters leads to an ability to create useful code, the resulting level of understanding will still fall short of our deeper goals. The programmer's view of a computer is that it is a magic box that efficiently executes commands; the internal mechanisms may remain a mystery. However, as scholars of computer science, we must know something of these mechanisms so that we can comprehend why a machine acts as it does, what its limitations are, and what improvements can be expected.

The second part of the book addresses the issue of how and why computers are able to compute.

Chapter 8 describes machine architecture and the organization of typical computers. It presents the basic hardware at the core of a computer system. Chapter 9 addresses the problem of translating a high-level computer language like Java into a lower-level language so that a program written in a high-level language can be run on a given architecture. Chapter 10 introduces concepts related to *operating systems*; these are the programs that bridge the gap between the user and the many hardware and software facilities on the machine. They make it easy for users to obtain the computing services that they want. Chapter 11 examines a topic of great concern in our networked world, computer security. As more and more of our lives become documented on machines and the connectivity of every machine to every other increases, we wonder if our lives will be secure in the new millennium. The final chapter of this section (12) introduces computer networks and the many concepts related to machines' talking to each other.

The final chapters of the book examine the limitations of computers and the frontiers of the science as it currently stands. Chapter 13 discusses problems related to program execution time and computations that require long processing times. Chapter 14 describes an attempt to speed up computers to take on larger problems, the introduction of parallel architectures. Chapter 15 discusses the existence of so-called noncomputable functions, and chapter 16 gives an introduction to the field of artificial intelligence.

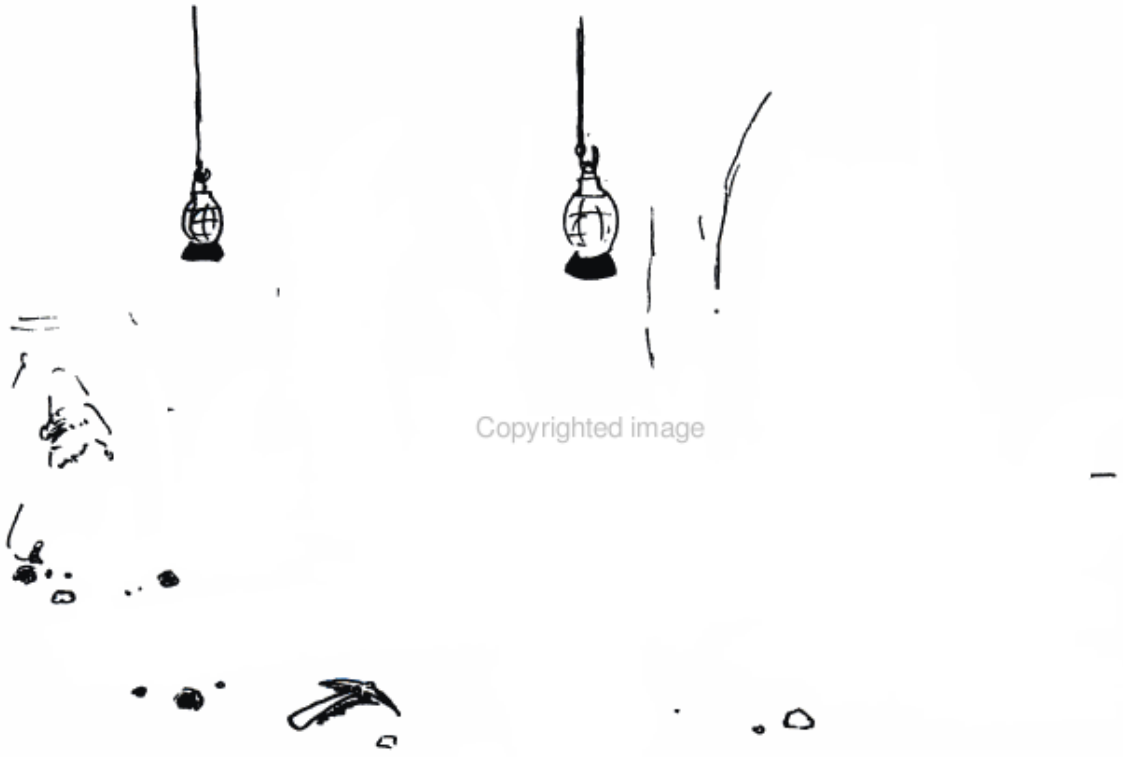
A great many programs have been developed to illustrate ideas in this book, and you can obtain them via the Internet. They can be found at Biermann's World Wide Web page at the Department of Computer Science, Duke University (http://www.cs.duke.edu/~awb).

An Approach for Nonmathematical Readers

A problem that arises in the teaching of computer science is that many instructors who know the field tend to speak in technical language and use too much mathematical notation for lay people to understand. Then the difficulty in communication leads them to conclude that ordinary people are not able to understand computer science. Thus, books and university courses often skirt the central issues of computer science and instead teach the operation of software packages or the history and sociology of computing.

This book was written on the assumption that intelligent people can understand every fundamental issue of computer science if preparation and explanation are adequate. No important topics have been omitted because of "difficulty." However, tremendous efforts were made to prune away unnecessary details from the topics covered and to remove special vocabulary except where careful and complete definitions are given.

Because casual readers may not wish to read every part of every chapter, the book is designed to encourage dabbling. Readers are encouraged to jump to any chapter at



1 The World Wide Web

World History and Where We Are

We begin by trying to decide what are the three most important events in human history. Which three occurrences since the beginning of time have had the greatest impact on the human species? Many might deserve this honor: the evolution of spoken language, the first use of tools, the discovery of fire, the discovery of the scientific method, the invention of the printing press, the Industrial Revolution, and so on. One could argue at some length about what the three greatest events have been. But this book suggests that a sure contender for the honor is the evolution of the *World Wide Web*, or more generally, a worldwide electronic communication network that potentially interconnects billions of individuals, businesses, governmental and educational institutions, libraries, and political, social, and religious groups as well as computers, databases, residences, automobiles, bicycles, briefcases, or even home appliances.

We know that the World Wide Web has been growing at an exponential rate. We know that businesses and governmental institutions have been rushing to get connected. We have seen dramatic steps in communication capabilities as fiber optic cables are stretched everywhere and earth satellites are deployed to keep us always in range. Almost every desk in every business has a personal computer, and we can now carry around laptop computers that communicate easily with the Web. We know we can find almost every book title in print, every newspaper being published, every airline flight that is scheduled, every movie title being shown, every university course being taught, and much more by simply clicking onto the Web and searching for the information. We know we can sit at home and do our job (at least in some cases or part of the time), participate in clubs, shop for clothes or a new car, sell specialized products, enter a chat room and share stories with complete strangers, explore a foreign land, or compete in games with people we have never met.

Where all of this will lead we do not know. It is both an exciting time and a frightening time in which to live. Of one thing we can be sure: it is a very good time to try to understand technology and especially networking. What is it, how does it work, what can

it do, what can it not do, how do we use it, and how do we stay safe within its environment? This chapter gets us started on these issues.

Let's Create Some Web Pages

We begin our study with an examination of the World Wide Web (WWW). We want to know what the World Wide Web is, and we want to utilize its resources to help us learn the many topics on our agenda. The task for this chapter is to learn to create Web pages and to connect them to the World Wide Web. Our later studies will build from these beginnings.

Here is text that we want to be able to view on a computer screen and that we want others on the World Wide Web to see on their screens also.

Alfred Nobel's Legacy

Alfred Nobel, a nineteenth-century industrialist, died in 1896 and left a will establishing the Nobel Foundation. This organization has been awarding, since 1901, annual prizes for outstanding accomplishments to scholars, literary figures, and humanitarian leaders.

The Nobel Prizes are currently given for contributions in six different areas of endeavor.

We give this page the title "The Nobel Prizes."

If we want this text to become a World Wide Web page, we must add formatting markers called *tags* to tell the computer display system how to present the text. The tags must be written in a language called Hyper Text Markup Language, or HTML. Here is the text with all the HTML tags included, to tell a computer system how to format the page.

```
<HTML>
<HEAD>
<TITLE> The Nobel Prizes </TITLE>
</HEAD>
<BODY>
<H1> Alfred Nobel's Legacy </H1>
<P> Alfred Nobel, a nineteenth-century industrialist, died
in 1896 and left a will establishing the Nobel Foundation.
This organization has been awarding, since 1901, annual
prizes for outstanding accomplishments to scholars,
literary figures, and humanitarian leaders. </P>
<P> The Nobel Prizes are currently given for contributions
in six different areas of endeavor. </P>
</BODY>
</HTML>
```

The markers are easy to understand. There is no deep rocket science here. Each tag is clearly identifiable by the angled brackets that surround it. Thus, if H1 is a tag name, <H1> is the tag. This tag refers to some text that begins at the point where <H1> appears and ends at the tag </H1>. Note that the end tags all have a slant (/) to denote that they mark an end.

Here are some examples of tags. The page title is presented as

```
<TITLE> The Nobel Prizes </TITLE>
```

and the heading of the paragraphs on this page is

```
<H1> Alfred Nobel's Legacy </H1>
```

The first paragraph of the text is typed as

```
<P> Alfred Nobel ... </P>
```

You, the designer of the Web page, are telling the computer display system how you want the page to look. Here is a list of the tags being used and their meanings:

<HTML>	The text surrounded by the tags <HTML> and </HTML> constitutes an HTML-formatted page that is to be displayed.
<HEAD>	The text surrounded by <HEAD> and </HEAD> gives heading information for this page.
<TITLE>	Part of the heading information is the page title. This title will be shown by the computer system near the page when it is displayed. But this title will not be on the page.
<BODY>	This tag defines the material to be on the displayed page.
<H1>	This tells the system to create a heading for the material.
<P>	This tells the system to create a paragraph.

In all these examples, every begin tag <M> has a complementary end tag </M>, indicating both the beginning and ending of the text being formatted. In later examples, some begin tags will not have complementary end tags because the endings will be obvious without them.

Having designed a page and typed the HTML tags to display it, we would now like to see the page displayed on a computer screen. The program that will display it is called a *browser*. The browser is especially designed to obey the HTML commands and to put the material we specify on the computer screen. (Two well-known browsers are Microsoft's Internet Explorer and Netscape Navigator.) We will type the HTML-formatted version into a computer file called `nobelinfo.html`. You should type this into your own computer using whatever editor that may be provided. It should be in a directory

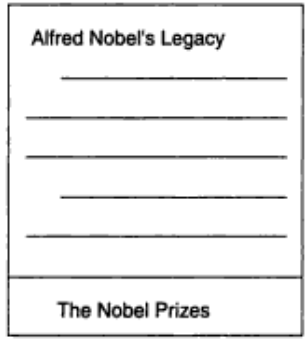


Figure 1.1

labeled `public_html` so that the browser can find it. Also, your computer must be attached to an Internet server. (You may need to get help from a friend to get started if you are not sure how to use your particular computer or if you do not know how to connect to an Internet server.)

Notice that when you use the browser to display your page, it will be formatted exactly as your HTML commands have stated but not necessarily as you have typed it. Thus if you typed two paragraphs but marked only one of them with the `<P>` and `</P>` tags, only one of them will be properly formatted by HTML. The browser follows the rules of HTML; it does not copy the way you have typed your text.

Thus, we have achieved a great thing. We have created a page and displayed it with a browser. This is the same browser that can reference pages from all over the world and display what other people wanted us to see. Those people have learned HTML and have used it to present their material to us. We are learning HTML so that we can return the favor. You can think of your page as an electronic entity sitting in the computer memory as shown in figure 1.1. Our next job will be to put more entities in that memory.

Now, let's create another page associated with the first, titled "Areas for Nobel Prizes." Here is the text for this page:

Nobel Prizes are given for outstanding contributions in these areas:

- Physics
- Chemistry
- Physiology or Medicine
- Literature
- Peace
- Economic Science

```
<HTML>
<HEAD>
<TITLE> Areas for Nobel Prizes </TITLE>
</HEAD>
<BODY>
Nobel Prizes are given for outstanding contributions in these areas:
<UL>
<LI> Physics
<LI> Chemistry
<LI> Physiology or Medicine
<LI> Literature
<LI> Peace
<LI> Economic Science
</UL>
<A HREF = "nobelinfo.html"> Return to main. </A>
</BODY>
</HTML>
```

Figure 1.4 shows how the two pages look with their installed links. You should type them in and make sure the links properly enable you to jump from one page to the other.

This is a nice result. But on more careful thought, it is much more than nice; it is an incredible, astounding, and world-altering discovery. The two pages compose a web—a small web, but still a web. And this is how the World Wide Web got started. Two pages of the kind we have made were linked together some years ago. Then more pages were added by various people and then more. Now there are tens of millions of people and pages all connected by the means we have shown, and they are changing the world in the manner we discussed in the first section.

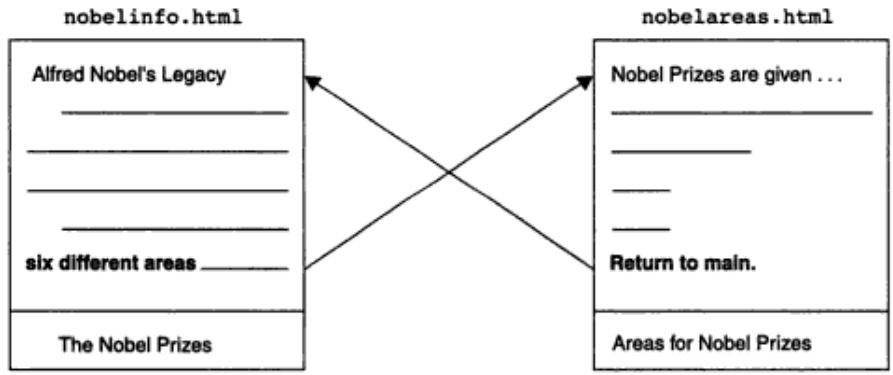


Figure 1.4

We can make still another change to our pages; we can put in an address to a remote site. Why not enable the user to click on the words "Nobel Foundation" and see the entry page (the *home page*) of the Nobel Foundation in Stockholm, Sweden? Not only will the user be able to see our pages but he or she will have a convenient way to jump to the source documents prepared by the originating organization. So here is another HTML version of the first page,

```
<HTML>
<HEAD>
<TITLE> The Nobel Prizes </TITLE>
</HEAD>
<BODY>
<H1> Alfred Nobel's Legacy </H1>
<P> Alfred Nobel, a nineteenth-century industrialist, died
in 1896 and left a will establishing the <A HREF =
"www.nobel.se"> Nobel Foundation </A>. This organization
has been awarding, since 1901, annual prizes for
outstanding accomplishments to scholars, literary figures,
and humanitarian leaders. </P>
<P> The Nobel Prizes are currently given for contributions
in <A HREF = "nobelareas.html"> six different areas </A> of endeavor.
</P>
</BODY>
</HTML>
```

Figure 1.5 shows a symbolic representation of the two pages with installed links.

Suddenly our toy classroom network has jumped to mammoth size because the Nobel Foundation page has numerous additional links for users to follow. A large number of pages that are interlinked in this way compose a *hypertext*; such entities have been studied by scientists for years. One can have arbitrarily many links from any page to any others. It contrasts with the standard notion of a book, which orders all its pages in a simple row. The psychological, artistic, and pedagogic implications of these two forms of organizing pages remain issues for research.

As an additional feature, the pages we have created are reachable by anyone on the Internet if you have placed them in the directory `public_html` and if your computer is connected to an Internet server. All that anyone needs to do is use a browser and go to the address of your *home site* adding `/nobelinfo.html` to the end of the address. For example, Biermann has a Web address of http://www.cs.duke.edu/~awb and The Nobel Prizes information page can be referenced by typing `http://www.cs.duke.edu/~awb/nobelinfo.html` into any Web-connected browser. (Remember that file `nobelinfo.html` was typed into directory `public_html` so that the browser could find it.)

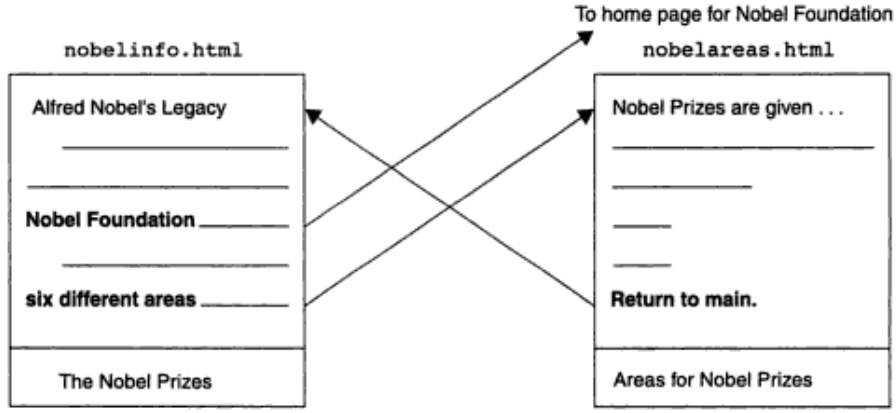


Figure 1.5

In order to make these links to distant pages actually work, the browsers that interpret the HTML and display the text must have a feature not yet discussed. If one has an `HREF` that points to another page, the browser must be able to follow that address and obtain the HTML document to display. This address-following capability is quite complex because it must send a request across communications networks to foreign computers and do the retrieval. There is an array of features that browsers must have to enable such references to succeed. They will not be discussed further here. We will only acknowledge that they exist, that they are complex, and that the browser must have them.

This concludes an ambitious section of this book. We examined a single page and how to display it with an ordinary browser. We showed how to construct links between pages and build a web or network. Finally, we connected our web to the World Wide Web, thus connecting ourselves to the dragon that is changing the world.

Exercises

1. Identify what, in your opinion, were the three most significant events in the history of humankind. Justify your answer.

2. Name a type of business (if you can) that will *not* be affected by the World Wide Web in the coming years. Justify your answer.

3. Type in the pages shown in this section and view them with a browser.

4. The first HTML page in this section has three different types of titles. Explain each one, its function, and where it appears on the display: The Nobel Prizes, Alfred Nobel's Legacy, and `nobelinfo.html`.

5. Create a home page for yourself using HTML, telling the major things that you would like the world to know about you. Have a friend view your Web page from a separate computer to be sure that the addressing conventions are working and that your page is truly on the Web.

6. Who won the Nobel Prize in Economics in 1982? What was the first year in which this prize was given? You should be able to answer these questions by following links from The Nobel Prizes page created in this section.

More HTML

This section describes a few more features of HTML. The reader should examine a standard reference for a complete description of this language and its many capabilities. (This section for the most part follows *A Beginner's Guide to HTML* as it appears at http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimer.html.)

In addition to the unordered list (`UL`) already described, HTML allows numbering of list items. This is done with the `<OL>` tag in the expected way:

```
<OL>
<LI> Physics
<LI> Chemistry
<LI> Economics
</OL>
```

which will appear on a Web page as a numbered list

1. Physics
2. Chemistry
3. Economics

Another characteristic of lists is that they can be nested. Thus, one can have a list of lists. We leave the exploration of this idea to the exercises.

One can create a list of definitions with the `<DL>` tag, as in the following:

```
<DL>
<DT> HTML
<DD> Hyper Text Markup Language
<DT> WWW
<DD> World Wide Web
</DL>
```

This will produce a two-column list on a Web page:

HTML
: Hyper Text Markup Language

WWW
: World Wide Web

In the previous section, `<H1>` was used to create a heading for a section of text. HTML has additional tags `<H2>`, `<H3>`, ..., `<H6>`, which create sequentially lower levels of headings. Thus, if we wanted to write a section for a Web page about HTML lists that contains subsections on unordered lists and ordered lists, we might type

```
<H1> Lists </H1>
<P> There are two types of lists, unordered lists and
ordered lists. </P>
<H2> Unordered Lists </H2>
<P> Unordered lists have the properties that ... </P>
<H2> Ordered Lists </H2>
<P> Ordered lists are also quite useful ... </P>
```

This would appear on the Web page as

Lists

There are two types of lists, unordered lists and ordered lists.

Unordered Lists

Unordered lists have the properties that ...

Ordered Lists

Ordered lists are also quite useful ...

To make words boldface or italic, use `<B>` or `<I>`.

Suppose you wish to type something and be guaranteed that the characters and spacing you use will be displayed on a Web page. You can use the tag `<PRE>` to do this. Thus, the HTML

```
<PRE>
     H   H     I
     H   H     I
     HHHHH     I
     H   H     I
     H   H     I
</PRE>
```

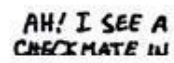
will show on a Web page as

2. Use Microsoft Word to create a document and have it generate your HTML automatically.

Summary

Observing the many changes in lifestyle and capabilities that the World Wide Web is enabling, we suspect that humankind is entering a new era. One cannot predict what wonderful things and what new dangers may present themselves in the coming years. But certainly we are wise to try to understand the World Wide Web so that we will be prepared to profit from its strengths and protect ourselves from its dangers. This chapter has provided a basis for that understanding by showing how to create a simple two-page web and how to make it part of the WWW.

We studied the basics of HTML and the use of a browser for viewing HTML pages. In the next chapter, we will learn that HTML is but a kindergarten-level introduction to what really can be done. Besides simply displaying a page on a screen, we might like to see lots of action on the page. Perhaps we would like the page to ask questions and respond to answers. Or maybe we would like the page to have buttons that can be pushed to activate actions of one kind or another. It is possible that we would like the page to calculate some useful numbers or to display little cartoon figures. All of this and more can be done with the programming language Java, as you will see in the next chapter.



2 Watch Out: Here Comes Java

Let's Put Some Action into Those Web Pages

In chapter 1, we studied how to create Web pages and link them to others on the World Wide Web. Now we want to put some action into these pages. We want them to jump around, flash bright colors, and compute complicated things. Java is a programming language that will enable us to do all these things and more.

As a first example, suppose we wish to modify our Nobel Prizes Web pages so that a user can ask for advice on finding some particularly interesting stories about Nobel prize winners. We assume the user will read the main page, and we place a question at the bottom of that page with buttons the user can push to indicate his answer. The question will be "Would you like to read about a scientist?" The user will be able to push the appropriate button, Yes or No, and then a new question will appear. By answering the series of questions, the user will get the desired advice. Here is a version of The Nobel Prizes page with the needed change:

```
<HTML>
<HEAD>
<TITLE> The Nobel Prizes </TITLE>
</HEAD>
<BODY>
<H1> Alfred Nobel's Legacy <H1/>
<P> Alfred Nobel, a nineteenth-century industrialist, died
in 1896 and left a will establishing the <A REF =
"www.nobel.se"> Nobel Foundation</A>. This organization
has been awarding, since 1901, annual prizes for
outstanding accomplishments to scholars, literary figures,
and humanitarian leaders. </P>
```

```
<P> The Nobel Prizes are currently given for contributions
in <A REF = "nobelareas.html"> six different areas </A> of
endeavor.
</P>
<P> If you would like to read an especially interesting
story about a prize winner, you should answer the questions
below for a suggestion. Then look up the suggested person
on the Nobel Foundation Web pages.
</P>
<APPLET code = "StoryAdvice.class"> </APPLET>
</BODY>
</HTML>
```

When this page is presented by the Web browser, the HTML code will cause it to jump at the step

```
<APPLET code = "StoryAdvice.class">
```

and to execute a program called an *applet*, which will do the work of asking the questions, placing the buttons on the screen, and responding to the button pushes. Here is how the page will appear, assuming the Java applet has been written carefully, placed in the same directory as the HTML file, and given the name `StoryAdvice.class`.

Alfred Nobel's Legacy

Alfred Nobel, a nineteenth-century industrialist, died in 1896 and left a will establishing the **Nobel Foundation**. This organization has been awarding, since 1901, annual prizes for outstanding accomplishments to scholars, literary figures, and humanitarian leaders.

The Nobel Prizes are currently given for contributions in **six different areas** of endeavor.

If you would like to read an especially interesting story about a prize winner, you should answer the questions below for a suggestion. Then look up the suggested person on the Nobel Foundation Web pages.

Would you like to read about a scientist?

Yes No

Advice:

Suppose the user pushes the Yes button. Then a new question will be asked: "Would you like to read about Albert Einstein?" and the user will be able to answer again. The questions will continue until the program comes to an appropriate point to present advice. Thus, if the user indicated an interest in a scientist and responded yes to the sug-

gestion of Albert Einstein, the program might respond that Einstein received the Nobel Prize in Physics in 1921 and that a brief biography of him is given on the Nobel Foundation Website.

The point is that HTML code can call a computer program that can be designed to execute almost any process we can imagine. Thus, our energies will now turn to the issue of computer programming—what it is and how you do it. That is the subject of this chapter and of several that follow it.

The Big Deal: Computer Programming

In the old days before computers, if we wanted to do a job, we had to *do* the job. But with computers, we can do many jobs by simply writing down what is to be done. A machine can do the work. If we want to add numbers, search for a fact, flash bright colors on a Web page, format and print a document, distribute messages to colleagues, or do other tasks, we can write a recipe for what is to be done and walk away while a machine obediently and tirelessly carries out our instructions. Our recipe could be distributed to many computers, and they could all work together to carry out our instructions. Even after we retire from this life, computers may still be employed to do the same jobs following the commands that we laid down.

The recipes that we are discussing will, of course, be coded as *programs*. The preparation and writing of them is called *programming*, which implements a kind of "work amplification" that is revolutionizing human society. Programming enables a single person to do a finite amount of work—the preparation of a computer program—and to achieve, with the help of a computer, an unbounded number of results. Thus, productivity is no longer simply a function of the number of people working; it is a function of the number of people and the number of machines working.

There is even more good news: computers are relatively inexpensive, and the cost to buy one is continually decreasing. Machines with 64,000-word memories and 1 microsecond instruction times cost $1 million four decades ago. Now we can buy a machine with roughly one thousand times the memory and speed for about $1,000. For the cost of one month of a laborer's time, we can purchase a machine that can do some tasks faster than a thousand people working together.

Object-Oriented Programming

To cash in on this obvious bonanza, we need to learn to program. But we want not only to program but to program well, and that leads to a long list of concerns that we consider in this book. Specifically, we want to be sure our programs are correct and that we

```
    public void SetStop(P x)
    {
        Java code
    }
    etc.
}
```

Another analogy might be a lawn mower. Data here consist of `Location`, `Direction`, `IsRunning` (true/false), `FuelLevel`, `OilLevel`, and `ThrottleSetting`. Some methods are `StartEngine`, `StopEngine`, `SetSpeed`, `GoForward`, `GoBackward`, `TurnLeft`, and `TurnRight`. The process of using the mower might involve executing some of these verbs (methods). One could go into more detail, but this is sufficient to illustrate a `LawnMower` class:

```
public class LawnMower
{
    Data Location, Direction, IsRunning, ... ;

    public void StartEngine()
    {
        Java code
    }
    public void StopEngine()
    {
        Java code
    }
    etc.
}
```

Exercises

1. Pick some nontrivial object and design a class for it. Note that it should include both data and functionality.
2. Data and function names may not be unique to a particular class. Think of a data item that might be suitable for both an electronic organ and a lawn mower.
3. Why would a method like `print` or `report` be useful with almost any class you can imagine?

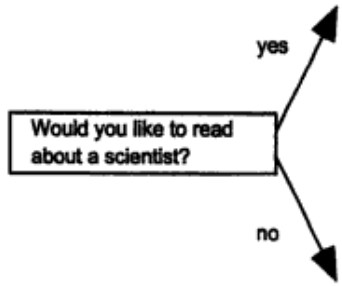


Figure 2.1

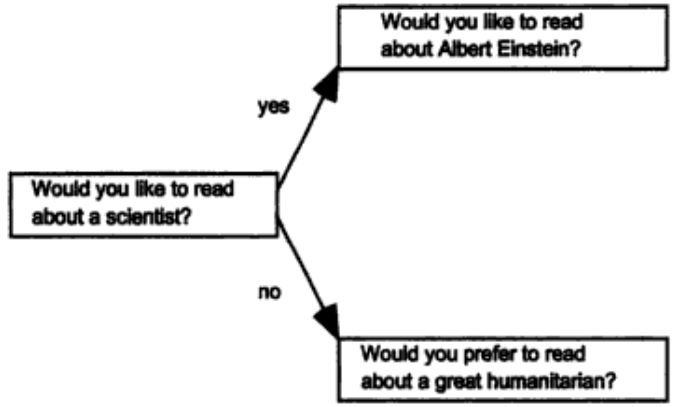


Figure 2.2

Following through the decision tree, we can trace a sample interaction. Assuming the user is interested in humanitarians, the path through the tree proceeds as follows:

```
Asking for Advice on an Interesting Story
Decision tree question: Would you like to read about a scientist?
Response: No
Decision tree question: Would you prefer to read about a great humanitarian?
Response: Yes
Decision tree advice: You might be interested in Aung San Suu Kyi, who won the Peace Prize in 1991.
```

This tree asks only two sequential questions before arriving at a decision. But it is easy to envision a large tree that asks many questions and recommends a wide variety of stories at the end of the path. It is also clear that this type of tree can be used to give advice on almost any subject from medical treatment to fortune-telling. Figure 2.4 shows an example of a medical advice decision tree.

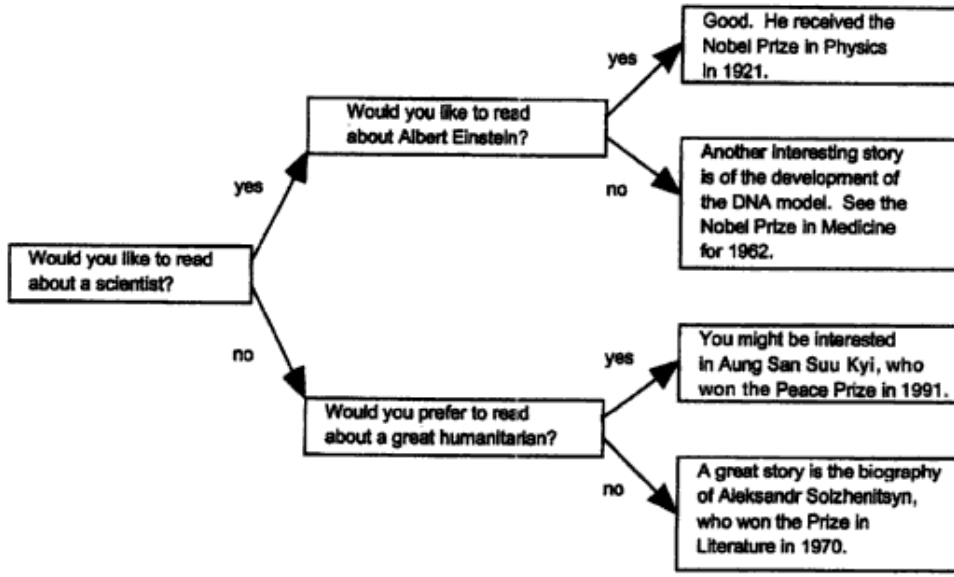


Figure 2.3

Another example is a game-playing tree. This can be illustrated by the simple game Nim, which has the following rules. The first player can place one, two, or three X's at the left end of a horizontal ladder; then the opponent can place one, two, or three O's in the next sequential squares. This chain of moves repeats again and again, filling the ladder from left to right, with the winner being the one to place a mark in the last square. An example for a ladder of seven squares is shown in figure 2.5. The first player might make three X's (fig. 2.5b). Then, suppose the second player makes two O's (fig. 2.5c). The first player could win by placing two more X's (fig. 2.5d).

Figure 2.6 shows a decision tree that will play the role of the second player for a Nim ladder of length 7.

Ordinarily we think of trees as emerging from the ground and spreading their branches toward the sky. The trees in this chapter move from left to right so they will be easier to program. The processing of the tree begins at the leftmost box, or *root node*, and proceeds along a path toward the right. At each decision *node*, the user of the tree is asked a question, and the answer given serves to select the next branch to be followed. The path proceeds toward the right until a final *leaf node* is encountered that has no outward branches. This leaf node will contain a message giving the result of the sequence of decisions, and it will terminate processing.

Such decision trees are applicable to a multitude of information-processing tasks, including advice giving, classification, instruction, and game-playing activities. Our task in this

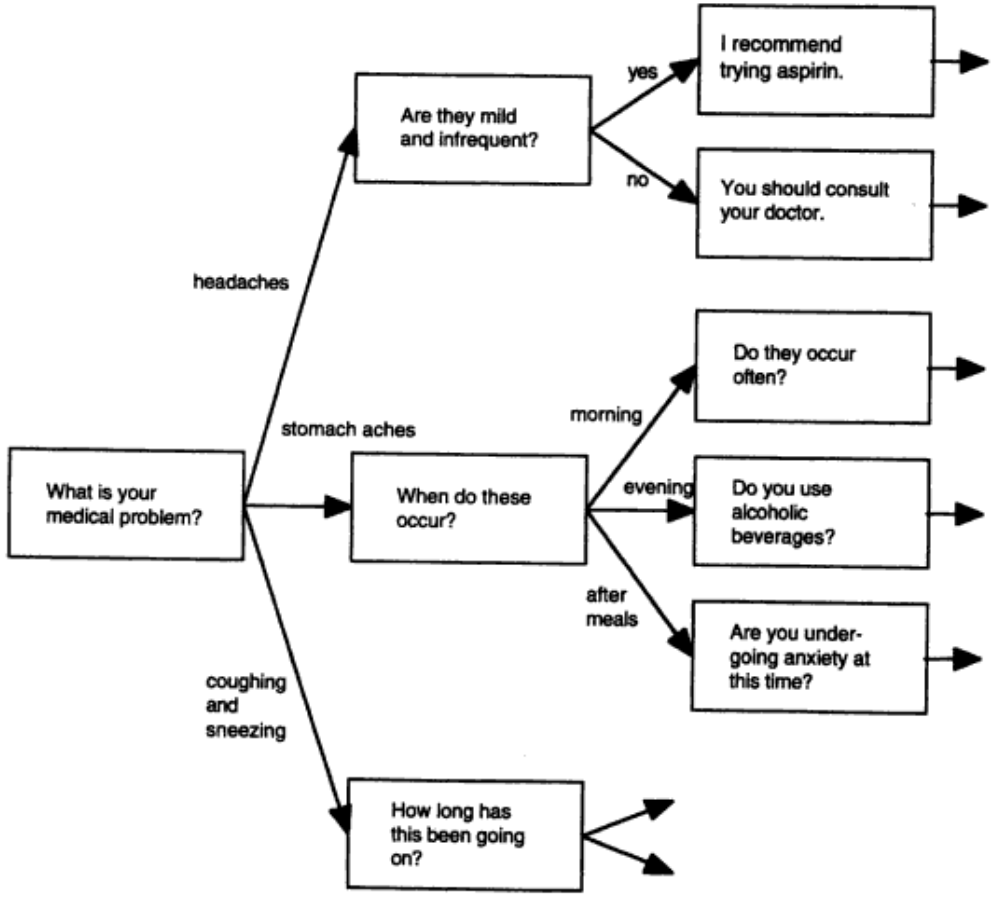


Figure 2.4

chapter is to write programs that contain such trees, lead a user down the correct paths, and print the results. The ability to design and program such trees is a powerful skill with many applications.

Reality Check

The discussion of decision trees showed an important part of the analysis required to program certain kinds of problems. Note that no mention of a computer or of a programming language was required. One implication is that a large part of computer programming is actually problem solving, that is, analyzing a problem and mapping out a

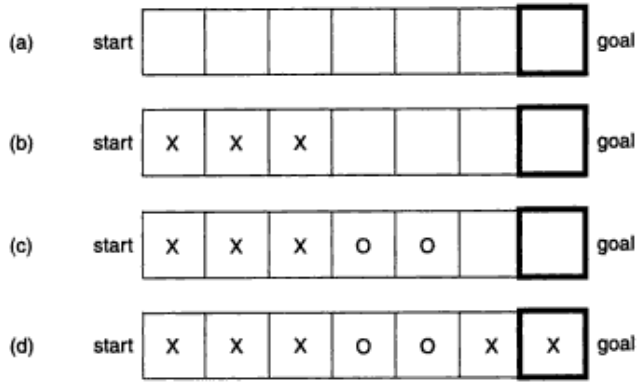


Figure 2.5

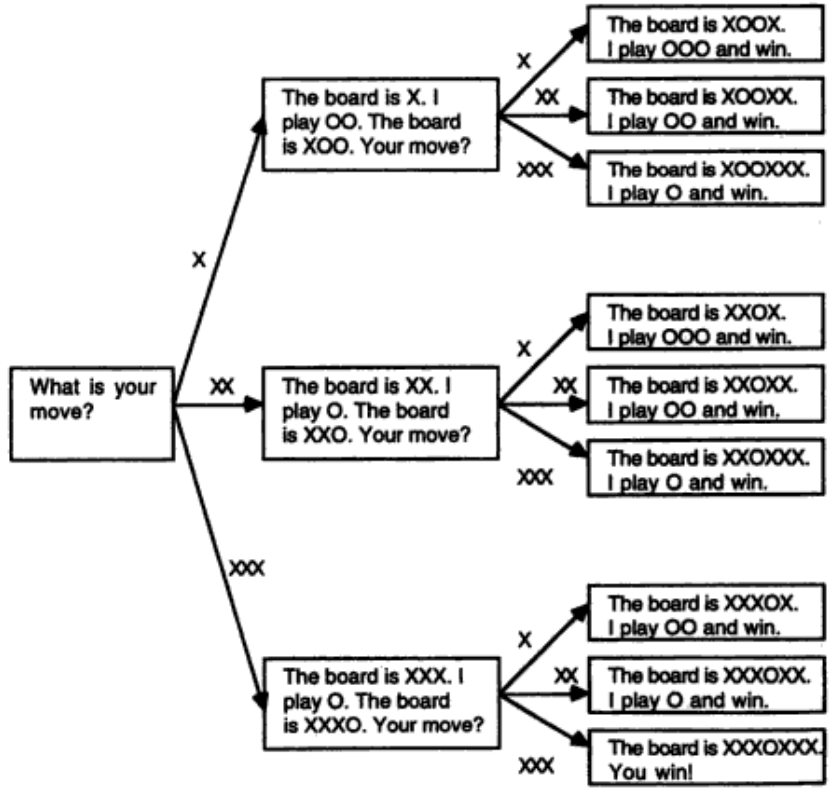


Figure 2.6

```
        m1.setText("H e l l o    W o r l d !");
        m2.setText("This is a simple Java test.");
        add(m1);
        add(m2);
    }
}
```

What does this applet do? It displays the following text on a Web page (the text will be inside rectangular boxes on the screen):

```
H e l l o    W o r l d !
```

This is a simple Java test.

Having the computer carry out the instructions in the program is called *running* or *executing* the program. The program will run on your computer only if you first translate the Java into executable form. Such translation is called *compiling*, and it can be done by calling a Java translator called a *compiler*. The usual way to do the translation is to use the command `javac`, but your machine may have a different way of doing this. Thus, for our example code, we must type

```
javac HelloWorld.java
```

The result is a file called `HelloWorld.class`, which will appear automatically in your current directory, presumably your `public_html` directory. This is the file that is executed by the browser from your HTML page. We study compilation in detail in chapter 9.

We use Java version 1.1 here, but later versions of Java should be compatible. However, if your browser processes only earlier versions of Java, the code shown here may not work. In order to properly face up to the issue of versions (or "dialects") and compilers, you may have to get some help in adapting to your specific situation. Computer science, like many disciplines, has an oral tradition, and some of the most important facts are passed on only by word of mouth. You may be able to get along without such help, but learning is usually easier and more fun if you can find it.

To get the maximum benefit from this chapter, you should type in and run some of the programs given here and observe their behaviors. For instance, we have told you how the previous example will print when run. Try out the program and see if you can get things to work the same way.

This program seems very simple, but many details related to its form and execution need to be understood. These include the composition of the program in terms of statements, the order of execution of the statements, and their meaning and structure. We will go through that in detail. However, look at the program and the output it produces. You can probably guess at what changes the program would require to have it print

`TextField` class and that we have named them `m1` and `m2`. Note that this statement, like most Java statements, ends with a semicolon.

After the data portion of the class, we see the definition of our only method. The method has its own header:

```
public void init()
```

The key point here is that the method is named `init` (every applet must have a method named `init`, for "initialize.") Normally the programmer gets to choose the names for the methods that are to be written, but this method is required. The body of this method is enclosed in opening and closing braces, as was the body of the whole class.

Our method declaration starts with the statements

```
m1 = new TextField(60);
m2 = new TextField(60);
```

This actually creates the `TextField` data objects `m1` and `m2` that we declared. The `new` causes the objects to be created. The `60` in the parentheses says to make the field large enough to hold 60 characters.

The lines

```
m1.setText("H e l l o     W o r l d !");
m2.setText("This is a simple Java test.");
```

use the `setText` method of the `TextField` class to put actual text into the `TextFields`. The text is enclosed in quotation marks and will appear exactly as typed in. Normally in Java the number of spaces is not important. Within quotation marks every space has significance. Note that the `m1.` on the first of these lines shows that `setText` is to be applied to `m1`, not `m2`. Similarly, `m2.` identifies the `TextField m2`.

The *object.verb*(*data*) syntax appears everywhere in Java programs, so you should become comfortable with it. It means do to *object* what *verb* says, using *data*. Returning to the organ example, suppose there are two declared organs, `or1` and `or2`. Suppose the method `PlayNote` needs one piece of data, the note to be played. Here is a sequence of Java commands that will play some notes on the two organs:

```
or1.PlayNote("C");
or1.PlayNote("D");
or1.PlayNote("E");
or2.PlayNote("A");
or1.PlayNote("F");
```

This sequence will play three sequential notes on the first organ `or1`, one note on `or2`, and then a last note on `or1`. Note that the verb must be appropriate for (defined for) the object being addressed. Thus we would not expect anything sensible to result from

```
orl.StartEngine();
```

because the verb `StartEngine` was defined for lawn mowers, not organs.

Our program will be a big disappointment if we do all this work and we never see the `TextFields`. We must tell the system to put them on the screen so we can see them. The last two statements do this. However, these statements do not say where to put the two `TextFields`. They will appear as two long rectangles that can contain text (up to 60 characters each, in this example.) The `add` statements give only minimal instructions as to where to put the `TextFields`. We specify that `m1` should come first, followed by `m2`. Note that this still leaves the browser a lot of freedom, and much depends on how big the display is. All we can specify with `add` is the order of placement, left to right and top to bottom, just as one would write English text on a piece of paper. (Java has more detailed ways of controlling the layout of objects on a screen. These are rather complicated, so most of them are omitted from this book.)

The first line of our example Java program,

```
import java.awt.*;
```

tells the system that we expect to include several optional Java features. For now, just use this as specified.

Exercises

1. The display after the example Java program suggested what the output of the program might look like with a typical browser. What might it look like with an extra-wide browser window?
2. What might it look like if the order of the `add` statements were reversed?

Program Execution

The computer functions by executing statements in order. It first finds the `init` method, and within that, it executes the first statement, the second, and so forth. It always executes statements in order unless special statements require it to do something else. Our example program is finished when it has executed the second `add` statement.

Statement Meaning and Structure

Having just gone through a simple Java program once, we need to go back and look at some things in more detail. The statement

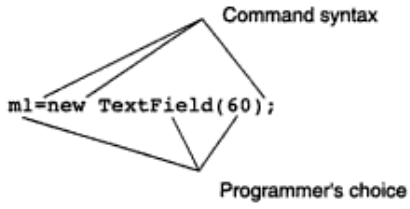


Figure 2.7

```
ml = new TextField(60);
```

has two parts (figure 2.7): the command syntax, which specifies exactly what ordering of characters is necessary for the command, and some data or keywords that the programmer has inserted. The command `new` tells the computer to create a new object named `ml`. The placement of the object name followed by the equals sign followed by the command `new` followed by the kind or type of object, with details, followed by a semicolon are all requirements for using the `new` command. This format is the syntax of the command. Items must be in exactly that order, and everything must be spelled correctly.

Three things are chosen by the programmer. The first is the object name, `ml`. Once chosen, it must be used consistently. The object type or class, here `TextField`, was also a choice of the programmer, but it was the only object provided by Java that would do what we required. Finally, the `60` was a choice of the programmer, specifying how big to make `TextField`. This `60` is essentially data to the statement.

The following version of the `init` method will not work correctly because of various errors:

```
Public Void Init()                                  // 1
{                                                   // 2
    ml = new TextField("sixty");                    // 3
    m2 new TextField(60);                           // 4
    ml.setText("H e l l o     W o r l d !);         // 5
    m2.setText("This is a simple Java test.")       // 6
    please add(ml);                                 // 7
    add m2;                                         // 8
}                                                   // 9
```

1. This line is wrong because the words each start with a capital letter rather than the correct form, which here is all lowercase letters.
2. The second line and ninth lines are correct, containing the opening and closing braces, respectively.

3. This line is wrong because it specifies the length of `TextField` with a string `"sixty"` instead of a number as required.
4. This line is missing the equals sign.
5. This line is missing the closing quotation mark between the exclamation point and the closing parenthesis.
6. This line is missing the statement-ending semicolon.
7. `please` seems polite but is syntactically incorrect.
8. This line is missing the parentheses around `m2`.

Most programming languages require perfect syntax, although a few allow some flexibility. Languages of the future may be less demanding.

If the command syntax is correct, the program will carry out the commands regardless of what is included as data. Thus, the following program will run, but maybe not as you hoped:

```
import java.awt.*;

public class HelloWorld extends java.applet.Applet
{
    TextField m1,m2;

    public void init()
    {
        m1 = new TextField(60);
        m2 = new TextField(60);
        m1.setText("C#7a-%% qwerty 496");
        m2.setText("Please ignore the nonsense above.");
        add(m2);
        add(m1);
    }
}
```

When this is run, we will get:

```
Please ignore the nonsense above.
C#7a-%% qwerty 496
```

Not only will it spew out the nonsense characters we typed in but since we have reversed the order of the `add` statements, the last remaining logic of the program has been destroyed because `above` should now be `below`.

The machine has no basis on which to judge the correctness of the data and will obediently carry out the instructions without regard to what is being manipulated. The computer will do precisely what you say even though that may not be what you want.

```
        lineA = new TextField(70);
        lineB = new TextField(70);
        lineC = new TextField(70);
        lineA.setText("**********************************");
        lineB.setText("Great Ideas in Computer Science");
        lineC.setText("**********************************");
        add(lineA);
        add(lineB);
        add(lineC);
    }
}
```

You should now be able to write a program that will print almost anything.

Exercises

1. If you were to write another applet similar to the `HelloWorld` applet, which word(s) in the header would you have to change?
2. What would happen to the output of the `HelloWorld` applet if the `add` statements were omitted?
3. What syntax errors can you identify in the following statement?

```
m2.setText("My favorite movie is "Star Wars," Episode 4");
```

Interactive Programs and Buttons

As you probably know from using a Web browser or other programs or computer games, many programs are *interactive*. That is, the program behavior—exactly what it does—depends on what you do with your mouse or type on your keyboard. In other words, the same program will not always do the same thing. What it does depends on what you do, what data you provide, what you click on, and so forth. An example is the Nobel Prize stories advice-giving program that we described previously.

One of the fundamental actions in operating a Web browser is using the mouse to place the cursor on a button displayed on a screen and pressing the left side of the mouse. This is called "clicking a button". Which button you click affects what the browser does and what it shows you. Since this is so important to most applets (including our example), the next programming example will show you how to create and utilize buttons.

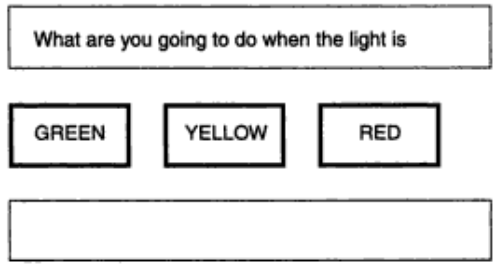


Figure 2.8

```
        if (cause == b3)
            {
            m2.setText("You must stop.");
            }
    }
}
```

Figure 2.8 shows what you should see when this program is first run. Then, if you press the GREEN button, the applet displays

Keep on rolling.

in the lower `TextField`. If you press the YELLOW button, the display is

Stop if you can!

in the `TextField`, and if you press the RED button, you see

You must stop.

With a high-level look at the code, you will see that we now have a second method. Before we had only `init`. For interactive applets, we require a method named `actionPerformed`. This new method is executed whenever an on-screen button is pressed with the mouse.

Before we track down the changes and additions to the rest of the program, let's look at this new method in detail. The method header,

```
public void actionPerformed(ActionEvent event)
```

is required and will be utilized unchanged in future programs. The body of the method is enclosed in braces and starts with

```
Object cause = event.getSource();
```

This statement assigns to the variable `cause` the identity of the object that causes this method to be executed. In this case, it tells which one of the three buttons was pressed. The remaining lines of the method use this information to place the correct message into the `TextField`:

```
if (cause == b1)
{
    m2.setText("Keep on rolling.");
}
if (cause == b2)
{
    m2.setText("Stop if you can!");
}
if (cause == b3)
{
    m2.setText("You must stop.");
}
```

We'll give the details for this kind of statement later, but you should find it plausible, from looking closely at this, that if button `b2`, the YELLOW button, is pressed, the message Stop if you can! is put into the `TextField m2`.

Let's move back to the `init` routine and see what changed there. Three `Buttons` and two `TextFields` are set up as in our first program:

```
m1 = new TextField(80);
m1.setText("What are you going to do when the light is");
b1 = new Button("GREEN");
b2 = new Button("YELLOW");
b3 = new Button("RED");
m2 = new TextField(80);
```

These are then positioned on the screen with a series of five `add` statements:

```
add(m1);
add(b1);
add(b2);
add(b3);
add(m2);
```

They tell the browser to first show the `TextField m1`, then follow that with the three `Buttons`, `b1`, `b2`, and `b3`. Note that these three buttons were coded to show GREEN, YELLOW, and RED, respectively. Last in the `add` series is the `TextField m2`.

What is totally new for the `init` method when writing an interactive applet is the following sequence of lines:

```
b1.addActionListener(this);
b2.addActionListener(this);
b3.addActionListener(this);
```

These use the `addActionListener` method of the `Button` class to register the three buttons so that the executing applet watches to see when they are pressed. If we had other buttons in our program but did not register them like this, then pressing one of those other buttons would have no effect.

There are two other modifications to our original program that need to be mentioned. The class header

```
public class TrafficLight extends java.applet.Applet
                    implements ActionListener
```

now includes `implements ActionListener` at the end. This tells Java that this class will deal with the actions directly. (This could have been dealt with in a separate class. That might be desirable in a much more complicated program.) In addition, we have had to add the line

```
import java.awt.event.*;
```

near the beginning. This tells Java we need to use the interactive features of the language.

Don't Panic

We have now looked at two Java programs in a fair amount of detail without trying to explain everything. Even so, the amount of syntax and detail may feel overwhelming. What is important here is that you understand the broad strokes. Even experienced programmers, when dealing with a new language, copy whole sections of code from a previous program unchanged and then make a few selected changes to "bend" the program to their needs.

When writing your first Java programs, you would do well to start by copying an example that does almost what you want your program to do. Get the example program working without making any of your modifications. Make your modification only when the example is working as was intended. That will ensure that you have copied everything correctly.

Later, we'll explain more of the items that we glossed over, and you will get used to using some of the Java features. Things will become much more comfortable than they are now.

Exercises

1. How would our `TrafficLight` program perform if we omitted the following line?

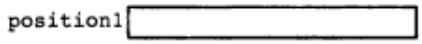
```
b2.addActionListener(this);
```

2. Write a Java program that answers a question about the country you live in. First, it should ask, "What would you like to know about *country name here*?" Then it should have buttons with labels such as "population," "total area," and so forth. When a user pushes one of the buttons, the answer will appear in a `TextField` provided.

Reading and Storing Data

In the previous section we introduced the concept of data—the information being manipulated by the program. We also showed how to set up buttons to select which data we wanted to print. Examples of data so far have been in the form of text strings such as `"Hello World"` and `"Stop if you can!"` You may have wondered if the program could print only data that had been put into the program by the programmer. The answer is no; the programmer does not have to anticipate all possible data a program might want to print and include it in the code. In this section we show how to get your program to collect such data from the keyboard: it will read the keystrokes from typed input and store the data in computer memory. This can be an important part of making a program interactive.

Before introducing the new statement to read data, it is necessary to talk about locations in memory. Such locations are like pigeonholes or containers with names where information can be stored and then retrieved when needed. For example, you might like to have a place in memory where a sequence of characters can be stored, and you might choose to name it `position1`. You could then store data in that location and use the data in various ways. You could instruct the machine to write the data into `position1` or to copy the data from `position1` to some other location.



The correct way to indicate in the Java language that such a memory location is to be set up is with a *declaration*. To store a collection of characters that we call a *string*, this declaration takes the form

```
String position1;
```

`position1` is a `String` variable and `getText` gets `String` information. (We have used assignments in our previous examples, to set up `Button` and `TextField`.)

Following is a simple example using the `getText` method and a `String` variable:

```
import java.awt.*;
import java.awt.event.*;
public class DupThree extends java.applet.Applet
                    implements ActionListener
{
    TextField m1, m2, m3, m4, m5;
    Button b1;
    String message;

    public void init ()
    {
        m1 = new TextField(80);
        m2 = new TextField(80);
        m3 = new TextField(80);
        m4 = new TextField(80);
        m5 = new TextField(80);
        b1 = new Button("button");
        m1.setText("Please enter some text below; then press button.");
        add(m1);
        add(m2);
        add(b1);
        add(m3);
        add(m4);
        add(m5);
        b1.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event)
    {
        message = m2.getText();
        m3.setText(message);
        m4.setText(message);
        m5.setText(message);
    }
}
```

Let's look at this in some detail. Our declarations

We then respond to this prompt and type something, say, `What a great day!` into the next `TextField`. Then we wait expectantly, but nothing happens. The computer has been told to watch for a press of `Button b1`. So we must press the button. This activates the `actionPerformed` method.

We now look at that portion of the program:

```
public void actionPerformed(ActionEvent event)
{
    message = m2.getText();
    m3.setText(message);
    m4.setText(message);
    m5.setText(message);
}
```

This is particularly simple because there is only one button and the program does not have to determine which button was pressed. We go right into the statement

```
message = m2.getText();
```

The `getText` method reads the message we typed into `TextField m2` and makes it available to the program. We assign this to the variable `message`. This means that whatever was typed into `TextField` now resides in a memory location named `message`, that is, the variable `message`. Now that this information is safely tucked away in the computer's memory, we can make use of it in subsequent statements. The statement

```
m3.setText(message);
```

employs the method `setText` and puts text into `TextField m3`, in effect printing it out.

Notice that two different forms of `setText` are employed. If quotation marks are used, the characters between the quotes are printed. Thus, with `m1.setText("Please enter ...");` the characters Please enter ... are printed (without quotation marks). We also specified the `String` variable `message`, however, and `m3.setText(message);` uses no quotation marks. It means treat `message` as a variable and print its contents. The contents of `message` are whatever was assigned to it in the previous statement. We had assumed that `What a great day!` had been typed in. This means `TextField m3` now contains What a great day! The next two statements,

```
m4.setText(message);
m5.setText(message);
```

do the same things for `TextFields m4` and `m5` (figure 2.10). In other words, whatever is typed into the program is duplicated three times; thus the name `DupThree` was chosen for the applet.

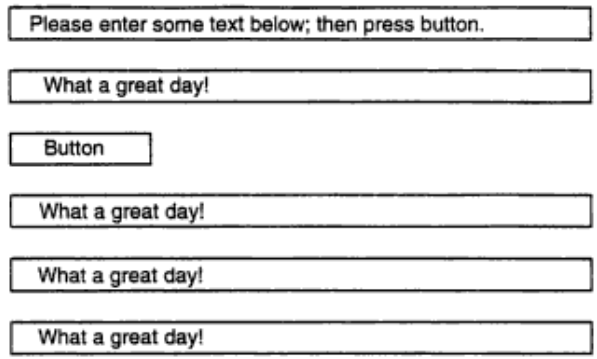


Figure 2.10

A few more points about variables are in order. The names of the places in memory, the variable names, can be almost anything as long as they begin with an alphabetic letter, include only alphabetic and numeric characters, and are properly declared. For example, the names `A17` and `c8Zi` could be used, but variable names like `taxe$`, `4sale`, and `short-cut` are invalid.

Names must follow two additional rules. First, they must not contain spaces. Second, there is a set of *reserved words* for the Java system that may not be used as names. Some examples are `if`, `import`, `public`, `class`, `extends`, `implements`, and `void`. These are all parts of the Java language, and we would confuse the system if we chose those as names. You might have to consult a Java manual for a complete list of reserved words.

An important point is that storage positions should not be used until something is loaded into them, as is done in the `DupThree` example by the assignment statements. Then, the variable will continue to hold that information unless it is replaced, possibly by some other assignment statement. If a variable is used at the left side of a subsequent assignment statement, whatever earlier information was contained in it is lost forever, replaced by the new information being assigned.

Exercises

1. Write Java statements to declare variables, using your choice of names, to store your name and address.

2. Design and write a Java program that prompts you for and reads in your first name and then your last name. Then have the program display them in the opposite order.

3. Design and write a Java program that is similar to the previous one but which contains two buttons labeled "First-First" and "Last-First." Design the program so that when the "First-First" button is pressed, the names are displayed in the input order. If the "Last-First" button is pressed, then the last name is displayed before the first name.

A Number-Guessing Game

In order to allow the ideas introduced in the previous section to "settle in," we'll go right to another program, which will enable us to play a game. The basic idea of the game is that one person selects a number between 1 and 100, and the other person tries to guess, as quickly as possible, what the "secret" number is. After each guess, the person with the secret number responds with "The secret is higher than the guess," "The secret is lower than the guess," or "That is the right answer." The program keeps track of everything and implements the details of the game.

The code is as follows:

```
import awb.*; // See Appendix for a discussion of this software.
import java.awt.*;
import java.awt.event.*;

public class AboveBelow extends java.applet.Applet
                    implements ActionListener
{
    TextField m1, m2;
    IntField i1; // This is not standard Java. See Appendix.
    Button b1, b2;
    int secret, guess;

    public void init ()
    {
        m1 = new TextField(80);
        m1.setText("Enter number between 0 and 100; then push
                    SECRET.");
        i1 = new IntField(40); // See Appendix.
        m2 = new TextField(80);
        b1 = new Button("SECRET");
        b2 = new Button("GUESS");
        add(m1);
        add(b1);
        add(i1);
        add(b2);
        add(m2);
```

```
        b1.addActionListener(this);
        b2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent event)
    {
        Object cause = event.getSource();

        if (cause == b1)
        {
            secret = i1.getInt();
            i1.setInt(); // extension of awb.*
            m1.setText("Now, enter your guess below; then press GUESS.");
        }
        if (cause == b2)
        {
            guess = i1.getInt();

            if (guess == secret)
            {
                m2.setText("You've got it!");
            }
            if (guess < secret)
            {
                i1.setInt();
                m2.setText("The number is greater than " + guess);
            }
            if (guess > secret)
            {
                i1.setInt();
                m2.setText("The number is less than " + guess);
            }
        }
    }
}
```

First, the big picture. We have a class named `AboveBelow` with some declarations in the body, followed by the required `init` method and then the `actionPerformed` method, which is required for any interactive applet. Our declarations include two `TextFields`, two `Buttons`, an `IntField`, and an `int`. The latter two are new. An `IntField` is a refinement of, and looks and behaves very much like, a `TextField` but is designed to deal with numeric data rather than strings. This is a special feature introduced by this book and

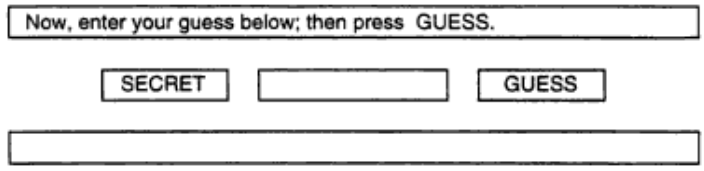


Figure 2.12

tity of the button. The first `if` will not find a match, since the GUESS button is `b2`. The following `if` statement does find a match:

```
if (cause == b2)
{
    guess = il.getInt();
```

This causes the assignment statement to get the number 50 that John entered into the `IntField`. So the memory location `guess` contains `50`. There are now three logical possibilities.

1. John hit it right on the money. The game is over.
2. John's guess is less than the secret number. The computer should tell him that so he can try again.
3. John's guess is greater than the secret number. The computer should report that and let him try again.

Here are the three `if` statements that accomplish this:

```
if (guess == secret)
{
    m2.setText("You've got it!");
}
if (guess < secret)
{
    il.setInt();
    m2.setText("The number is greater than " + guess);
}
if (guess > secret)
{
    il.setInt();
    m2.setText("The number is less than " + guess);
}
```

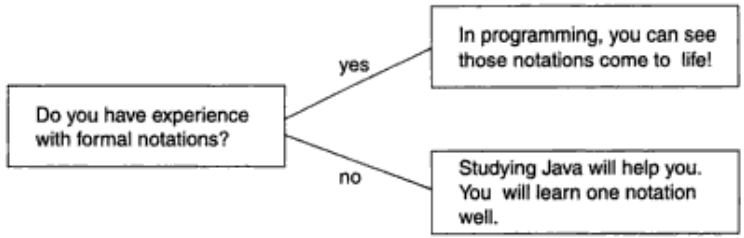


Figure 2.14

Without any strategy, for example, random guessing, it could take up to 100 tries. With a binary search strategy, what is the maximum number of guesses it would take if the secret number were between 1 and 1,000?)

Exercises

1. Come up with another guessing strategy for the number game. Is it better or worse than binary search?
2. Design a similar game program, using words rather than numbers. The secret and the guesses would all be strings. Describe how the program would work, but do not attempt to write a program.
3. What guessing strategy might you use for a word-oriented game?
4. Compare how you use a telephone book to the strategy you came up with in exercise 3.

Programming Decision Trees

Let's begin by programming the simplest possible tree, one with only one branching node (figure 2.14). We studied in the previous sections how to make the machine print these kinds of messages and how to have it read the user's answers. We also introduced, without serious explanation, the `if` statement, which allowed us to do or not do something depending on the answer to the `if` clause. We need to explore the `if` statement and its `if-else` variant more carefully.

In general, the `if-else` statement has the following form:

```
if (some true/false expression)
{
    Java code A
```

```
}
else
{
    Java code B
}
```

In actual code, the true/false expression might be something like `cause == b1`, which we saw in a previous program. It is asking if `cause` is equal to `b1` or, in effect, is the button that was pressed `Button b1`? The answer to that is either true or false (yes or no). The way the `if-else` statement works is that if the outcome of the question is true, then Java code A is executed. If the outcome is false, then Java code B is carried out instead. Here Java code A and Java code B are sequences of Java statements. These sequences with their enclosing braces are called compound statements, which are more carefully defined later.

A simplified variant, called the `if` statement, is just like the `if-else` statement but drops the "else" clause. In this case, if the result of the true/false question is true, the statements right after the `if` are executed. If the result is false, nothing is executed. The form is

```
if (some true/false expression)
{
    Java code A
}
```

We are now able to write the computer program for this simple decision tree:

```
import java.awt.*;
import java.awt.event.*;

public class SimpTree extends java.applet.Applet
                    implements ActionListener
{
    TextField mQuery, mAnswer;
    Button bYes, bNo;

    public void init()
    {
        mQuery = new TextField(70);
        mQuery.setText("Do you have experience with formal notations?");
        bYes = new Button("Yes");
        bNo = new Button("No");
```

```
        mAnswer = new TextField(70);
        bYes.addActionListener(this);
        bNo.addActionListener(this);
        add(mQuery);
        add(bYes);
        add(bNo);
        add(mAnswer);
    }

    public void actionPerformed(ActionEvent event)
    {
        Object cause = event.getSource();
        if (cause == bYes)
        {
                    mAnswer.setText("In programming, you can see ...");
        }
        else // must have been the No button
        {
                    mAnswer.setText("Studying Java will help you ...");
        }
    }
}
```

As you can see, the solution for this ultrasimple case is almost identical to our `TrafficLight` program except that we only have two buttons. Let's go through this code:

```
import java.awt.*;
import java.awt.event.*;

public class SimpTree extends java.applet.Applet
                    implements ActionListener
{
    TextField mQuery, mAnswer;
    Button bYes, bNo;
```

We use the standard import statements and then have a class header that names the class `SimpTree`, says it is a special case of an applet, and states that the "listener" for any buttons is implemented in this class. Then we declare the `TextFields` `mQuery` and `mAnswer` and the `Buttons` `bYes` and `bNo`.

Next is the `init` method that initializes things, that is, it sets up everything so that it is ready to go. Only the body of `init` is shown in the following:

```
mQuery = new TextField(70);
mQuery.setText("Do you have experience with formal notations?");
bYes = new Button("Yes");
bNo = new Button("No");
mAnswer = new TextField(70);
bYes.addActionListener(this);
bNo.addActionListener(this);
add(mQuery);
add(bYes);
add(bNo);
add(mAnswer);
```

The first two lines create and put text into the `mQuery TextField`. The next two statements create buttons labeled Yes and No. Next we create the `TextField` where we will eventually place our "answer." The `addActionListener` statements register the two buttons to be listened for. Finally the four `add` statements place on the screen the query `TextField` first, then the two `Buttons`, and then the `TextField` that will contain the answer.

When this applet is run, the `init` method will result in a display such as the one shown in figure 2.15. Pressing one of the two buttons, either Yes or No, will cause the `actionPerformed` method to be started up. Let's look at the body of that routine:

```
Object cause = event.getSource();
if (cause == bYes)
{
    mAnswer.setText
    ("In programming, you can see those notations come to life!");
```

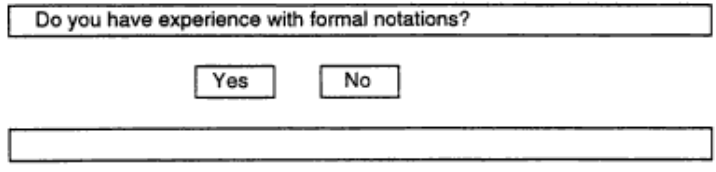


Figure 2.15

We'll tackle more complicated decision trees later, but first we need to deal with some points of grammar for Java programming.

Grammar and Style

By now you have probably noticed that (almost) all statements are terminated by a semicolon (;). Let's systematically go through the various kinds of statements we have encountered. Statements invoking methods for various classes end with semicolons. Examples are

```
mQuery.setText("Do you have experience with formal notations?");
```

and

```
add(mQuery);
```

Note also that when invoking a method, parentheses, ( ), are always included after the name of the method. There may or may not be something between the parentheses. Declarations are statements ending with semicolons. For example,

```
Button bYes, bNo;
```

Assignment statements, which always include an equals sign (=) end with a semicolon, as shown in the next example:

```
mAnswer = new TextField(70);
```

The exception, so far, is the `if` statement, which itself is not ended in a semicolon but normally includes one or more statements within the braces that are terminated by semicolons. Method and applet definitions consist of a header and then statements enclosed in braces. The closing brace for such definitions is not followed by a semicolon.

We also spoke earlier of compound statements. These are just collections of one or more statements enclosed in braces. The compound statement itself is not terminated in a semicolon but the statements inside are.

The example programs you have seen so far also follow fairly rigid style rules, that is, the formatting, specifically the indentation, is done carefully and systematically. Just like an outline of a report, indentation reflects the logical structure of a program. As we stated earlier, the distribution of spaces and statements on a line is usually of little consequence to the Java compiler. But it is extremely important to human readers. Since it is very important that programmers fully understand the programs they are working with, any formatting that improves readability is, in practice, a big benefit.

For now, the indentation rules can be summarized by saying that all lines within braces are indented. With every opening brace, the degree of indentation moves one unit to the

```
{
    TextField mQuery, mAnswer;
    Button bYes, bNo;
    int myLocation;

    public void init()
    {
        mQuery = new TextField(70);
        mQuery.setText("Would you like to read about a scientist?");
        bYes = new Button("Yes");
        bNo = new Button("No");
        myLocation = 0;
        mAnswer = new TextField(70);
        bYes.addActionListener(this);
        bNo.addActionListener(this);
        add(mQuery);
        add(bYes);
        add(bNo);
        add(mAnswer);
    }

    public void actionPerformed(ActionEvent event)
    {
        Object cause = event.getSource();
        if (myLocation == 0)
        {
            if (cause == bYes)
            {
                myLocation = 1;
                mQuery.setText("Would you like to read about Einstein?");
            }
            if (cause == bNo)
            {
                myLocation = 2;
                mQuery.setText("Would you prefer a humanitarian?");
            }
        }
        else if (myLocation == 1)
        {
```

```
            if (cause == bYes)
            {
                myLocation = 3;
                mAnswer.setText("He received the Physics Prize in 1921.");
            }
            if (cause == bNo)
            {
                myLocation = 4;
                mAnswer.setText("See the Prize in Medicine, 1962.");
            }
        }
        else if (myLocation == 2)
        {
            if (cause == bYes)
            {
                myLocation = 5;
                mAnswer.setText("Look up the Peace Prize, 1991.");
            }
            if (cause == bNo)
            {
                myLocation = 6;
                mAnswer.setText("Try the Literature Prize, 1970.");
            }
        }
    }
}
```

Notice that the class header, declaration, and the `init` method are almost unchanged from our previous decision tree implementation. The additional declaration statement

```
int myLocation;
```

sets up an integer variable that will help us keep track of where in the decision tree we have last been. That variable will utilize the node numbers shown in figure 2.18. The two *different* lines in the `init` method are

```
mQuery.setText("Would you like to read about a scientist?");
```

```
myLocation = 0;
```

Since the question here is different, the `setText` statement contains the new question. We also set the value of `myLocation` to zero to show that we have been at the root node. This

```
<statement> ==> <name> = <name>;
```

Finally, we apply the rule that says <name> can be replaced by a sequence of characters beginning with a letter.

```
<statement> ==> <name> = address;
```

Or the last step could have been done differently to obtain

```
<statement> ==> <name> = "Hello, my name is Oscar.";
```

This assumes we have a variable named address and that someone would be interested in inserting "Hello, my name is Oscar." into a program. To complete the process, we need to replace <name> in each case using the rule that says a name is a string of alphanumeric symbols that begins with a letter. So, the following is consistent:

```
<statement> ==> heading = address;
<statement> ==> message = "Hello, my name is Oscar.";
```

Again it assumes that both heading and message are valid string variable names.

Take a Deep Breath ...

Let's step back and see what we have accomplished. We have a series of rules that collectively describe (a portion of) the Java language syntax. We started with a syntactical element, in this case a <statement> and then by a series of substitutions ended up with two syntactically correct Java statements. We did that with very simple substitutions and by selecting rules that matched our needs. Wherever we had something in angle brackets, <>, we looked for a rule that allowed us to replace that item either with a more detailed description using the angle bracket notation or with actual code. When we were done, we had produced actual code for statements.

So, in addition to developing a notation that describes the syntax of Java, we have illustrated a *method* for generating *syntactically correct* Java statements.

If this seems a bit arbitrary and random, it is, because we were just trying to illustrate the process. Normally, you have a Java statement or program for which you want to verify the syntax. That is what the Java compiler does when compiling your programs, and it lets you know when it is unhappy with your syntax.

More Rules

To fully illustrate this technique, we need to spell out a few more rules and then verify some actual code. For now, we will skip much of what pertains to integers and arithmetic. Also, since we have already presented rules that include the items <class> and <string-method>, we will defer detailed examination of these.