

Chapter 1

What is Programming Anyway?

Most of the machines that have been developed to improve our lives serve a single purpose. Just try to drive to the store in your washing machine or vacuum the living room with your car and this becomes quite clear. Your computer, by contrast, serves many functions. In the office or library, you may find it invaluable as a word processor. Once you get home, slip in a DVD and your computer takes on the role of a television. Start up a flight simulator and it assumes the properties of anything from a hang glider to a Learjet. Launch an mp3 player and you suddenly have a music system. This, of course, is just a short sample of the functions a typical personal computer can perform. Clearly, the computer is a very flexible device.

While the computer's ability to switch from one role to another is itself amazing, it is even more startling that these transformations occur without making major physical changes to the machine. Every computer system includes both hardware, the physical circuitry of which the machine is constructed, and software, the programs that determine how the machine will behave. Everything described above can be accomplished by changing the software used without changing the machine's actual circuitry in any way. In fact, the changes that occur when you switch to a new program are often greater than those you achieve by changing a computer's hardware. If you install more memory or a faster network card, the computer will still do pretty much the same things it did before but a bit faster (hopefully!). On the other hand, by downloading a new application program through your web browser, you can make it possible for your computer to perform completely new functions.

Software clearly plays a central role in the amazing success of computer technology. Very few computer users, however, have a clear understanding of what software really is. This book provides an introduction to the design and construction of computer software in the programming language Java. By learning to program in Java, you will acquire a useful skill that will enable you to construct software of your own or participate in the implementation or maintenance of commercial software. More importantly, you will gain a clear understanding of what a program really is and how it is possible to radically change the behavior of a computer by constructing a new program.

A program is a set of instructions that a computer follows. We can therefore learn a good bit about computer programs by examining the ways in which instructions written for humans resemble and differ from computer programs. In this chapter we will consider several examples of instructions for humans in order to provide you with a rudimentary understanding of the nature of a computer program. We will then build on this understanding by presenting a very simple but complete example of a computer program written in Java. Like instructions for humans, the instructions

that make up a computer program must be communicated to the computer in a language that it comprehends. Java is such a language. We will discuss the mechanics of actually communicating the text of a Java program to a computer so that it can follow the instructions contained in the program. Finally, you have undoubtedly already discovered that programs don't always do what you expect them to do. When someone else's program misbehaves, you can complain. When this happens with a program you wrote yourself, you will have to figure out how to change the instructions to eliminate the problem. To prepare you for this task, we will conclude this chapter by identifying several categories of errors that can be made when writing a program.

1.1 Without Understanding

You have certainly had the experience of following instructions of one sort or another. Electronic devices from computers to cameras come with thick manuals of instructions. Forms, whether they be tax forms or the answer sheet for an SAT exam, come with instructions explaining how they should be completed. You can undoubtedly easily think of many other examples of instructions you have had to follow.

If you have had to follow instructions, it is likely that you have also complained about the quality of the instructions. The most common complaint is probably that the instructions take too long to read. This, however, may have more to do with our impatience than the quality of the instructions. A more serious complaint is that instructions are often unclear and hard to understand.

It seems obvious that instructions are more likely to be followed correctly if they are easy to understand. This "obvious" fact, however, does not generalize to the types of instructions that make up computer programs. A computer is just a machine. Understanding is something humans do, but not something machines do. How can a computer understand the instructions in a computer program? The simple answer is that it cannot. As a result, the instructions that make up a computer program have to satisfy a very challenging requirement. It must be possible to follow them correctly without actually understanding them.

This may seem like a preposterous idea. How can you follow instructions if you don't understand them? Fortunately, there are a few examples of instructions for humans that are deliberately designed so that they can be followed without understanding. Examining such instructions will give you a bit of insight into how a computer must follow the instructions in a computer program.

First, consider the "mathematical puzzle" described below. To appreciate this example, don't just read the instructions. Follow them as you read them.

1. Pick a number between 1 and 40.
2. Subtract 20 from the number you picked.
3. Multiply by 3.
4. Square the result.
5. Add up the individual digits of the result.
6. If the sum of the digits is even, divide by 2.
7. If the result is less than 5 add 5, otherwise subtract 4.

8. Multiply by 2.
9. Subtract 6.
10. Find the letter whose position in the alphabet is equal to the number you have obtained (a=1, b=2, c=3, etc.)
11. Think of a country whose name begins with this letter.
12. Think of a large mammal whose name begins with the second letter of the country's name.

You have probably seen puzzles like this before. The whole point of such puzzles is that you are supposed to be surprised that it is possible to predict the final result produced even though you are allowed to make random choices at some points in the process. In particular, this puzzle is designed to leave you thinking about elephants. Were you thinking about an elephant when you finished? Are you surprised we could predict this?

The underlying reason for such surprise is that the instructions are designed to be followed without being understood. The person following the instructions thinks that the choices they get to make in the process (choosing a number or choosing any country whose name begins with "D"), could lead to many different results. A person who understands the instructions realizes this is an illusion.

To understand why almost everyone who follows the instructions above will end up thinking about elephants, you have to identify a number of properties of the operations performed. The steps that tell you to multiply by 3 and square the result ensure that after these steps the number you are working with will be a multiple of nine. When you add up the digits of any number that is a multiple of nine, the sum will also be a multiple of nine. Furthermore, the fact that your initial number was relatively small (less than 40), implies that the multiple of nine you end up with is also relatively small. In fact, the only possible values you can get when you sum the digits are 0, 9 and 18. The next three steps are designed to turn any of these three values into a 4 leading you to the letter "D". The last step is the only point in these instructions where something could go wrong. The person following them actually has a choice at this point. There are several countries whose names begin with "D" including Denmark, Djibouti, Dominica and the Dominican Republic. Luckily, for most readers of this text, Denmark is more likely to come to mind than any of the other countries (even though the Dominican Republic is actually larger in both land mass and population).

This example should make it clear that it is possible to *follow* instructions without understanding how they work. It is equally clear that it is not possible to *write* instructions like those above without understanding how they work. This contrast provides an important insight into the relationship between a computer, a computer program and the author of the program. A computer follows the instructions in a program the way you followed the instructions above. It can comprehend and complete each step individually but has no understanding of the overall purpose of the program, the relationships between the steps, or the ways in which these relationships ensure that the program will accomplish its overall purpose. The author of a program, on the other hand, must understand its overall purpose and ensure that the steps specified will accomplish this purpose.

Instructions like this are important enough to deserve a name. We call a set of instructions designed to accomplish some specific purpose even when followed by a human or computer that has no understanding of their purpose an *algorithm*.

There are situations where specifying an algorithm that accomplishes some purpose can actually be useful rather than merely amusing. To illustrate this, consider the standard procedure called long division. A sample of the application of the long division procedure to compute the quotient $13042144/32$ is shown below:

$$\begin{array}{r}
 407567 \\
 32 \overline{)13042144} \\
 \underline{128} \\
 242 \\
 \underline{224} \\
 181 \\
 \underline{160} \\
 214 \\
 \underline{192} \\
 224 \\
 \underline{224} \\
 0
 \end{array}$$

Although you may be rusty at it by now, you were taught the algorithm for long division sometime in elementary school. The person teaching you might have tried to help you understand why the procedure works, but ultimately you were probably simply taught to perform the process by rote. After doing enough practice problems, most people reach a point where they can perform long division but can't even precisely describe the rules they are following, let alone explain why they work. Again, this process was designed so that a human can perform the steps without understanding exactly why they work. Here, the motivation is not to surprise anyone. The value of the division algorithm is that it enables people to perform division without having to devote their mental energies to thinking about why the process works.

Finally, to demonstrate that algorithms don't always have to involve arithmetic, let's consider another example where the motivation for designing the instructions is to provide a pleasant surprise. Well before you learned the long division algorithm, you were probably occasionally entertained by the process of completing a connect-the-dots drawing like the one shown in Figure 1.1. Go ahead! It's your book. Connect the dots and complete the picture.

A connect-the-dots drawing is basically a set of instructions that enable you to draw a picture without understanding what it is you are actually drawing. Just as it wasn't clear that the arithmetic you were told to perform in our first example would lead you to think of elephants, it is not obvious looking at Figure 1.1 that you are looking at instructions for drawing an elephant. Nevertheless, by following the instructions "Connect the dots" you will do just that (even if you never saw an elephant before).

This example illustrates a truth of which all potential programmers should be aware. It is harder to devise an algorithm to accomplish a given goal than it is to simply accomplish the goal. The goal of the connect-the-dots puzzle shown in Figure 1.1 is to draw an elephant. In order to construct this puzzle, you first have to learn to draw an elephant without the help of the dots. Only after you have figured out how to draw an elephant in the first place will you be able to figure out where to place the dots and how to number them. Worse yet, figuring out how to place and number the dots so the desired picture can be drawn without ever having to lift your pencil from the paper can be tricky. If all you really wanted in the first place was a picture of an elephant, it



Figure 1.1: Connect dots 1 through 82 (©2000 Monkeying Around)

would be easier to draw one yourself. Similarly, if you have a division problem to solve (and you don't have a calculator handy) it is easier to do the division yourself than to try to teach the long division algorithm to someone who doesn't already know it so that they can solve the problem for you.

As you learn to program, you will see this pattern repeated frequently. Learning to convert your own knowledge of how to perform a task into a set of instructions so precise that they can be followed by a computer can be quite challenging. Developing this ability, however, is the key to learning how to program. Fortunately, you will find that as you acquire the ability to turn an informal understanding of how to solve a class of problems into a precise algorithm, you will be developing mental skills you will find valuable in many other areas.

1.2 The Java Programming Language

An algorithm starts as an idea in one person's mind. To become effective, it must be communicated to other people or to a computer. Communicating an algorithm requires the use of a language. A program is just an algorithm expressed in a language that a computer can comprehend.

The choice of the language in which an algorithm is expressed is important. The numeric calculation puzzle that led you to think of Danish elephants was expressed in English. If our instructions had been written in Danish, most readers of this text would not understand them.

The use of language in a connect-the-dots puzzle may not be quite as obvious. Note, however, that we could easily replace the numbers shown with numbers expressed using the roman numerals I, II, III, IV, ... LXXXII. Most of you probably understand Roman numerals, so you would still be able to complete the puzzle. You would probably have difficulty, however, if we switched to something more ancient like the numeric system used by the Babylonians or to something more modern like the binary system that most computers use internally, in which the first few dots would be labeled 1, 10, 11, 100, 101, and 110.

The use of language in the connect-the-dots example is interesting from our point of view because the language used is quite simple. Human languages like English and Japanese are very complex. It would be very difficult to build a computer that could understand a complete human language. Instead, computers can only interpret instructions written in simpler languages designed specifically for computers. Computer languages are much more expressive than a system for writing numbers like the Roman numerals, but much simpler in structure than human languages.

One consequence of the relative simplicity of computer languages is that it is possible to write a program to translate instructions written in one computer language into another computer language. These programs are called *compilers*. The internal circuitry of a computer usually can only interpret commands written in a single language. The existence of compilers, however, makes it possible to write programs for a single machine using many different languages. Suppose that you have to work with a computer that can understand instructions written in language A but you want to write a program for the machine in language B. All you have to do is find (or write) a program written in language A that can translate instructions written in language B into equivalent instructions in language A. This program would be called a compiler for B. You can then write your programs in language B and use the compiler for B to translate the programs you write into language A so the computer can comprehend the instructions.

Each computer language has its own advantages and disadvantages. Some are simpler than others. This makes it easier to construct compilers that process programs written in these languages.

At the same time, a language that is simple can limit the ways you can express yourself, making it more difficult to describe an algorithm. Think again about the process of constructing the elephant connect-the-dot puzzle. It is easier to draw an elephant if you let yourself use curved lines than if you restrict yourself to straight lines. To describe an elephant in the language of connect-the-dot puzzles, however, you have to find a way to use only straight lines. On the other hand, a language that is too complex can be difficult to learn and use.

In this text, we will teach you how to use a language named Java to write programs. Java provides some sophisticated features that support an approach to programming called object-oriented programming that we emphasize in our presentation. While it is not a simple language, it is one of the simpler languages that support object-oriented programming.

Java is a relatively young computer language. It was designed in the early 90's by a group at Sun Microsystems. Despite its youth, Java is widely used. Compilers for Java are readily available for almost all computer systems. We will talk more about Java compilers and how you will use them once we have explained enough about Java itself to let you write simple programs.

Our approach to programming includes an emphasis on what is known as *event-driven programming*. In this approach, programs are designed to react to *events* generated by the user or system. The programs that you are used to using on computers use the event-driven approach. You do something – press a button, select an item from a menu, etc. – and the computer reacts to the “event” generated by that action. In the early days of computing, programs were started with a collection of data all provided at once and then run to completion. Many text books still teach that approach to programming. In this text we take the more intuitive event-driven approach to programming. Java is one of the first languages to make this easy to do as a standard part of the language.

1.3 Your First Sip of Java

The task of learning any new language can be broken down into at least two parts: studying the language's rules of grammar and learning its vocabulary. This is true whether the language is a foreign language, such as French or Japanese, or a computer programming language, such as Java. In the case of a programming language, the vocabulary you must learn consists primarily of verbs that can be used to command the computer to do things like “**show** the number 47.2 on the screen” or “**send** the next email message to the server.” The grammatical structures of a computer language enable you to form phrases that instruct the computer to perform several primitive commands in sequence or to choose among several primitive commands based on a user's input.

When learning a new human language, one undertakes the tasks of learning vocabulary and grammar simultaneously. One must know at least a little vocabulary before one can understand examples of grammatical structure. On the other hand, developing an extensive vocabulary without any knowledge of the grammar used to combine words would just be silly. The same applies to learning a programming language. Accordingly, we will begin your introduction to Java by presenting a few sample programs that illustrate fundamentals of the grammatical structure of Java programs, using only enough vocabulary to enable us to produce somewhat interesting examples.

1.3.1 Simple Responsive Programs

The typical program run on a personal computer reacts to a large collection of actions the user can perform using the mouse and keyboard. Selecting menu items, typing in file names, pressing



Figure 1.2: Window displayed by a very simple program

buttons, and dragging items across the screen all produce appropriate reactions from such programs. The details of how a computer responds to a particular user action are determined by the instructions that make up the program running on the computer. The examples presented in this section are intended to illustrate how this is done in a Java program.

To start things off simply, we will restrict our attention to programs that involve only one form of user input mechanism: buttons. The programs we consider in this section will only specify how the computer should respond when the user clicks on a button in the program's window.

As a very simple first example, consider the structure of a program which simply displays some text on the screen each time a button is clicked. When this program is started, all that will appear in its window is a single button as shown in Figure 1.2. The window remains this way until the user positions the mouse cursor within the button and presses the mouse button. Once this happens, the program displays the phrase

I'm Touched

in the window as shown in Figure 1.3. If the user clicks the mouse again, the program displays a second copy of the phrase "I'm Touched" in the window. Eventually, if the user clicks often enough, the whole window will be filled with copies of this phrase. That is all it does! Not exactly Microsoft Word, but it is sufficient to illustrate the basic structure of many of the programs we will discuss.

The text of such a Java program is shown in Figure 1.4. A brief examination of the text of the program reveals features that are certainly consistent with our description of this program's behavior. There is the line

```
contentPane.add( new JLabel( "I'm Touched" ) );
```

which includes the word "add" and the message to be displayed. This line comes shortly after a line containing the words "button clicked" (all forced together to form the single word `buttonClicked`)



Figure 1.3: Window displayed by a very simple program

```
import squint.*;
import javax.swing.*;

public class TouchyButton extends GUIManager {

    private final int WINDOW_WIDTH = 150;
    private final int WINDOW_HEIGHT = 300;

    public TouchyButton() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );
        contentPane.add( new JButton( "Click Here" ) );
    }

    public void buttonClicked() {
        contentPane.add( new JLabel( "I'm Touched" ) );
    }

}
```

Figure 1.4: Our first Java program

which suggest when the new message will appear. Similarly, a few lines earlier, a line containing the word `createWindow` is followed by a line containing the words `new JButton`. These two lines, describe actions that should happen when the program initially begins to run. These suggestive tidbits are unfortunately obscured by a considerable amount of text that is probably indecipherable to the novice. Our goal is to guide you through the details of this program in a way that will enable you to understand its basic structure.

1.3.2 “Class” and other Magic Words

Our brief example program contains many words that have special meaning to Java. Unfortunately, it is relatively hard to give a precise explanation of many of the terms used in Java to someone who is just beginning to program. For example, to fully appreciate the roles of the terms `import`, `public` and `extends` one needs to appreciate the issues that arise when constructing programs that are orders of magnitude larger than we will discuss in the early chapters of this text. We will attempt here to give you some intuition regarding the purpose of these words. However, you may not be able to understand them completely until you learn more about Java. Until then, we can assure you that you will do fine if you are willing to regard just a few of these words and phrases as magical incantations that must be recited appropriately at certain points in your program. For example, the first two lines of nearly every program you read or write while studying this book will be identical to the first two lines in this example:

```
import squint.*;
import javax.swing.*;
```

In fact, these two lines are so standard that we won't even show them in the examples beyond the first two chapters of this text.

The Head of the Class

Most of your programs will also contain a line very similar to the third line shown in our example:

```
public class TouchyButton extends GUIManager {
```

This line is called a *class header*. The programs you write will all contain a line that looks just like this except that you will replace the word `TouchyButton` with a word of your own choosing. `TouchyButton` is just the name we have chosen to give to our program. It is appropriate to give a program a name that reflects its behavior.

This line is called a class header because it informs the computer that the text that follows describes a new `class`. Why does Java call the specification that describes a program a “class”? Java uses the word `class` to refer to:

“A set, collection, group, or configuration containing members regarded as having certain attributes or traits in common.” (From the American Heritage Dictionary)

If several people were to run the program shown above at the same time but on different computers, each would have an independent copy of the program described by this `class`. If one person clicked on the button, the message “I’m Touched” would only appear on that person’s computer. The other computers running the same program would be unaffected. Thus, the running copies of the program are independent but they form a collection very similar objects. Java refers to such a collection of objects as a *class*.

Using Software Libraries

The class header of `TouchyButton` indicates that it **extends** something called `GUIManager`. This means that our program depends on previously written Java instructions.

Programs are rarely built from scratch. The physical circuits of which a computer is constructed are only capable of performing very simple operations like changing the color of a single dot on the screen. If every program was built from scratch, every program would have to explicitly describe every one of the primitive operations required to accomplish its purpose. Instead, libraries have been written containing collections of instructions describing useful common operations like drawing a button on the screen. Programs can then be constructed using the operations described by the library in addition to the operations that can be performed by the basic hardware.

This notion of using collections of previously written Java instructions to simplify the construction of new programs explains the mysterious phrases found in the first two lines of our program. Lines that start with the words `import` inform Java which libraries of previously written instructions our program uses. In our example, we list two libraries, `javax.swing` and `squint`. The library named `javax.swing` is a collection of instructions describing common operations for building programs that use interface mechanisms like buttons and menus. The prefix “`javax`” reveals that this library is a standard component of the Java language environment used by many Java programs.

The second library mentioned in our `import` specifications is `Squint`. This is a library designed just for this text to make the Java language more appropriate as an environment for teaching programming. Recall that the class header of our example program mentions that `TouchyButton` extends `GUIManager`. `GUIManager` refers to a collection of Java instructions that form part of this `Squint` library. A `GUIManager` is an object that coordinates user and program activities that involve Graphical User Interface (GUI) components like buttons, menus, and scrollbars. If a program was nothing but a `GUIManager`, then all that would happen when it was run would be that a window would appear on the screen. Nothing would ever appear within the window. Our `GUIManager` class specification extends the functionality of the `GUIManager` by telling it exactly what GUI components to display (a button and “labels” that display the message “I’m Touched”) and when they should be displayed.

Getting Braces

The single open brace (“{”) that appears at the end of the class header for `TouchyButton` introduces an important and widely used feature in Java’s grammatical structure. Placing a pair consisting of an open and closing brace around a portion of the text of a program is Java’s way of letting the programmer indicate that the enclosed text describes a single, logical unit. If you scan quickly over the complete example, you will see that braces are used in this way in several parts of this program even though it is quite short.

The open brace after the class header

```
public class TouchyButton...
```

is matched by the closing brace on the last line of the program. This indicates that everything between these two braces (i.e., everything left in the example) should be considered part of the description of the class named `TouchyButton`. The text between these braces is called the *body* of the class.

Within the body of this class, there are two other examples of structures composed of a header line followed by a body that is enclosed in curly braces. The first begins with the header line

```
public TouchyButton() {
```

and the second begins with the header

```
public void buttonClicked() {
```

Like the class header, both of these lines start with the word **public**. Unlike the class header, these lines don't include the word **class** or the phrase **extends GUIManager**. Finally, both of these headers end with a name followed by a pair of empty parentheses. The names included before the parentheses reveal that these lines are the headers of similar but distinct grammatical structures in Java. The first is the header of a structure called a *constructor*. It specifies the actions to be performed when this program is first started. We can tell this because the name included in this header is the same as the name of the class we are defining, **TouchyButton**. The second is the header of a construct called a *method definition*. We discuss these two constructs in the following sections.

1.3.3 Under Construction

In order to appreciate the role of a constructor, one has to appreciate Java's notion that a class doesn't describe a single object, but instead describes a whole collection of similar objects. Given a blueprint for a house, you can build multiple copies of the house. Similarly, given a class definition, you "build" a new copy of the program defined each time you run the program. Just as you have to build a house to live in it, you must *construct* a copy or *instance* of a class definition before you can interact with it. This is what happens when you actually run a program described by a class definition. The role of the constructor within a class definition is to specify the details of the steps that should be performed to construct an instance of a class definition.

With this background, let us take a close look at the constructor included in the definition of the **TouchyButton** class:

```
public Touchybutton() {
    this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );
    contentPane.add( new JButton( "Click Here" ) );
}
```

The header of this constructor is quite simple. The key elements are the word **public**, the name of the class in which the constructor appears, and a set of empty parentheses. Eventually, we will learn that it is possible to provide additional information between the parentheses, but for at least a few chapters, all of our constructor headers will take this simple form.

The interesting part of the constructor is its body. The body of a constructor contains a list of instructions telling the computer what operations to perform when it is told to create a new instance of the object described by its class definition. In our constructor for the **TouchyWindow** class we include the two instructions:

```
this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );
contentPane.add( new JButton( "Click Here" ) );
```

These two instructions tell the computer that when this program begins to execute it should create a new window on the screen for this program's use and then display a button in that window.

Both of the instructions included in this constructor use a notation we will be seeing quite frequently. Both instructions have the general form

```
name.action( ... );
```

This form of instruction is used to tell some component of the program, identified by the name that appears before the period, to perform the action described after the period. It is called a *method invocation*. In the first method invocation the name is `this` and the action is `createWindow`. The name `this` refers to the program itself. It may seem odd, but in this command, the program is telling itself to create a new window. In the second instruction, the name is `contentPane` and the action is `add`. The name `contentPane` refers to the interior of the program window — the part that holds the window's contents. This second command tells the content pane that it should add a new button to the items it displays.

In each of the method invocations used in this constructor, extra information that is needed to specify how to perform the desired action is provided between the parentheses that appear after the action name. These extra pieces of information are called the *arguments* or *actual parameters* of the method invocation. For example, in the first line of the constructor the arguments included determine the size of the window we are instructing the program to create.

Obviously, by changing the contents of these instructions, we can change the way in which this program behaves. For example, if we replaced the second instruction in the constructor's body with the line:

```
contentPane.add( new JButton( "Click Me" ) );
```

the button displayed in the window would contain the words "Click Me" instead of "Click Here". Accordingly, you cannot simply view the instructions placed within a constructor as magical incantations. Instead, you must carefully consider each component so that you understand its purpose.

1.3.4 Discourse on the Method

The last few lines in the body of the class `TouchyButton` look like:

```
public void buttonClicked() {
    contentPane.add( new JLabel( "I'm Touched" ) );
}
```

This text is an example of another important grammatical form in Java, the *method definition*. A method is a named sequence of program instructions. A method definition consists of a *method header*, which includes the name to be associated with the instructions, followed by a *method body* (which is bracketed by braces just like the body of the constructor). In this case, the method being defined is named `buttonClicked`, and its body contains the single instruction:

```
contentPane.add( new JLabel( "I'm Touched" ) );
```

In the method header, the method's name is preceded by the words `public void` and followed by an empty pair of parentheses. Unfortunately, for now, you must just trust that these words and parentheses are necessary. They will appear in all of the methods we examine in this chapter. On

the other hand, a programmer is free to choose any appropriate name for a method. The method name can then be used in other parts of the program to cause the computer to obey the instructions within the method's body.

Within a class that extends `GUIManager`, certain method names have special significance. In particular, if such a class contains a method which is named `buttonClicked`

then the instructions in that method's body will be followed by the computer whenever any button displayed in the program's window is pressed using the mouse. That is why this particular program reacts to a button click as it does.

The single line that forms the body of our `buttonClicked` method:

```
contentPane.add( new JLabel( "I'm Touched" ) );
```

is another example of a method invocation. It specifies that a new "label" displaying the phrase

```
I'm Touched
```

should be added to the `contentPane` associated with the program.

In addition to `buttonClicked`, other special method names (`menuItemSelected`, for example) can be used to specify how to react to other actions the user may perform when interacting with a program. All such methods are called *event-handling methods*. We will provide a complete list of such methods later.

1.3.5 The Name Game

The constructor definition in this program is preceded by the lines

```
private final int WINDOW_WIDTH = 150;
private final int WINDOW_HEIGHT = 300;
```

These two lines are not instructions that the computer will follow when it needs to react to some event. They are definitions that tell the computer what meanings should be associated with the names `WINDOW_WIDTH` and `WINDOW_HEIGHT` throughout this program's execution.

We could have written this program without defining these names. If we replaced the first line in the constructor with the line

```
this.createWindow( 150, 300 );
```

we could remove the two definitions that precede the constructor and the program would still behave in the same way. We used the names `WINDOW_WIDTH` and `WINDOW_HEIGHT` for two reasons. First, their use illustrates good programming style. When writing a program it is not only important to give the computer the correct instructions, it is also important to write the instructions in a way that will make it as easy as possible for someone else to read and understand them. Even as a beginner, it is pretty easy to guess that the command

```
this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );
```

has something to do with setting the program's window size. This isn't as obvious with the command

```
this.createWindow( 150, 300 );
```

As one learns to write large, complicated programs, taking care to write them in a style that will be as easy to understand as possible becomes very important.

Our other reason for using these names is just to give you a hint about how important names are in computer programs. We have already seen that names like `this` and `ContentPane` are used to refer to elements of your program. The meanings associated with these names are built into Java and the libraries we are using. Names like `WINDOW_WIDTH` and `WINDOW_HEIGHT` are interesting because they show that it is also possible for you as a programmer to associate meanings with names. This important ability is the main topic of the next chapter.

1.4 Programming Tools

Writing a program isn't enough. You also have to get the program into your computer and convince your computer to follow the instructions it contains.

A computer program like the one shown in the preceding section is just a fragment of text. You already know ways to get other forms of textual information into a computer. You use a word processor to write papers. When entering the body of an email message you use an email application like Eudora or Outlook. Just as there are computer applications designed to allow you to enter these forms of text, there are applications designed to enable you to enter the text of a program.

Entering the text of your program is only the first step. As explained earlier, unless you write your program in the language that the machine's circuits were designed to interpret, you need to use a compiler to translate your instructions into a language the machine can comprehend. Finally, after this translation is complete you still need to somehow tell the computer to treat the file(s) created by the translation process as instructions and to follow them.

Typically, the support needed to accomplish all three of these steps is incorporated into a single application called an *integrated development environment* or IDE. It is also possible to provide separate applications to support each step. Which approach you use will likely depend on the facilities available to you and the inclination of the instructor teaching you to program. There are too many possibilities for us to attempt to cover them all in this text. To provide you with a sense of what to expect, however, we will sketch how two common integrated development environments, BlueJ and Eclipse, could be used to enter and run the `TouchyButton` program. These sketches are not intended to provide you with the detailed knowledge required to actually use either of these IDEs effectively. We will merely outline the main steps that are involved.

The IDEs we will describe share several important properties:

- Implementations of both IDEs are available for a wide range of computer systems including Windows systems, MacOS, and Unix and its variants.
- Both IDEs are available for free and can be downloaded from the web.

They also differ in major ways. BlueJ was designed by computer science faculty members with the primary goal of providing a Java development system for use when teaching students to program. Eclipse was developed to meet the needs of professional programmers.

Just as it is helpful to divide a book into chapters, it is helpful to divide large programs into separate text files describing individual components of the program. Within a Java IDE, the collection of text files that constitute a program is called a project. In fact, most Java IDEs expect all programs, even programs that are quite small, to be organized as projects. As a result, even

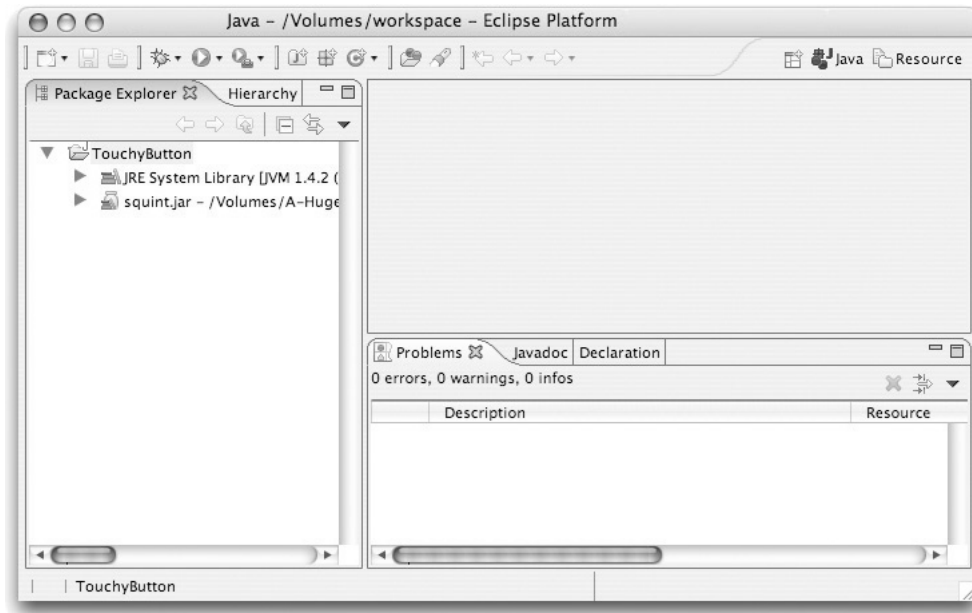


Figure 1.5: An Eclipse project window

though our `TouchyButton` program is only 13 lines long and will definitely only require one text file, the first step performed to enter and run this program using either Eclipse or BlueJ will be to use an entry in the application’s “File” menu to create a new project.

The IDE will then display a number of dialog boxes asking for information about the project we wish to create. Among other things, we will be asked to specify a name for the project and to select a location in our computer’s file system to store the file(s) that will hold the text of our program.

Once a project has been created, the IDE will display a window representing the state of the project. The window Eclipse would present is shown in Figure 1.5 and the window BlueJ would present is shown in Figure 1.6.

The next step is to tell the IDE that we wish to create a new file and enter the text of our program. With BlueJ, we do this by pressing the “New Class...” button seen in the upper left of the window shown in Figure 1.6. With Eclipse, we select a “New Class” menu item. In either case, the IDE will then ask us to enter information about the class, including its name, in a dialog box. Then the IDE will present us with a window in which we can type the text of our program much as we would type within a word processor. Such program text is often called *code*.

In Figures 1.7 and 1.8 we show how the windows provided by Eclipse and BlueJ would look after we ask the IDE to create a new class and enter the code for the class. Eclipse incorporates the text entry window as a subwindow of the project window. BlueJ displays a separate text window.

In both the BlueJ project window and the BlueJ window holding the text of the `TouchyButton` class there is a button labeled “Compile”. Pressing this button instructs BlueJ that we have completed entering our code and would like to have it translated into a form the machine can more easily interpret. Under most Java IDEs, compiling a class definition will produce a file storing a translation of the definition into a language called Java virtual machine code or byte code. After

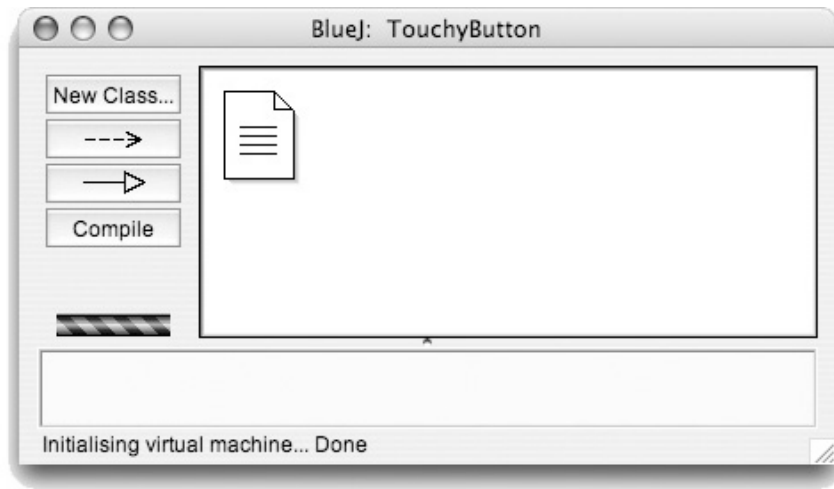


Figure 1.6: A BlueJ project window

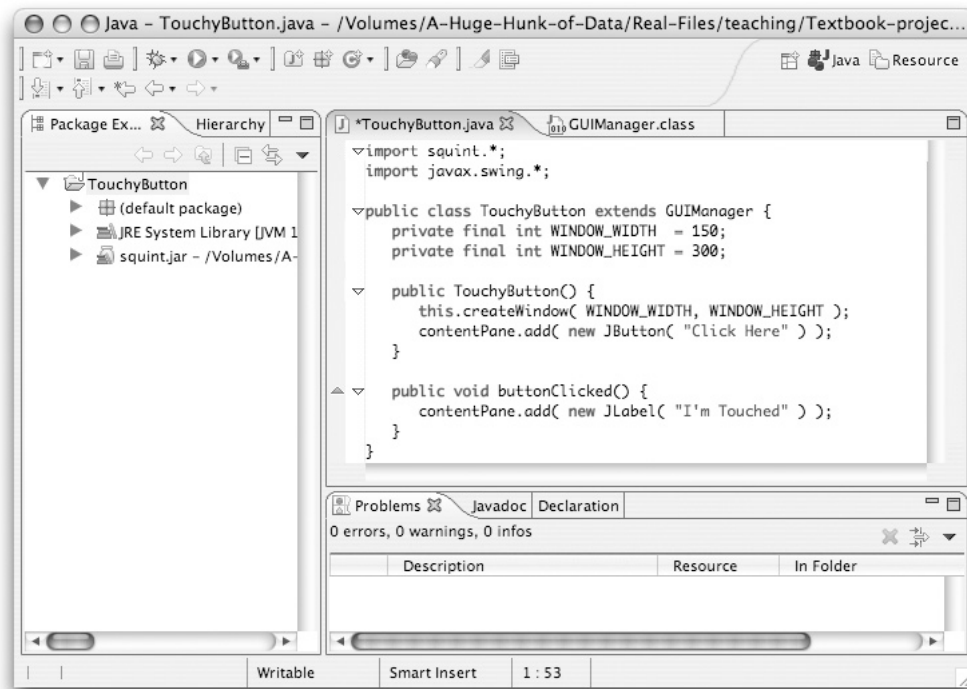


Figure 1.7: Entering the text of TouchyButton under Eclipse

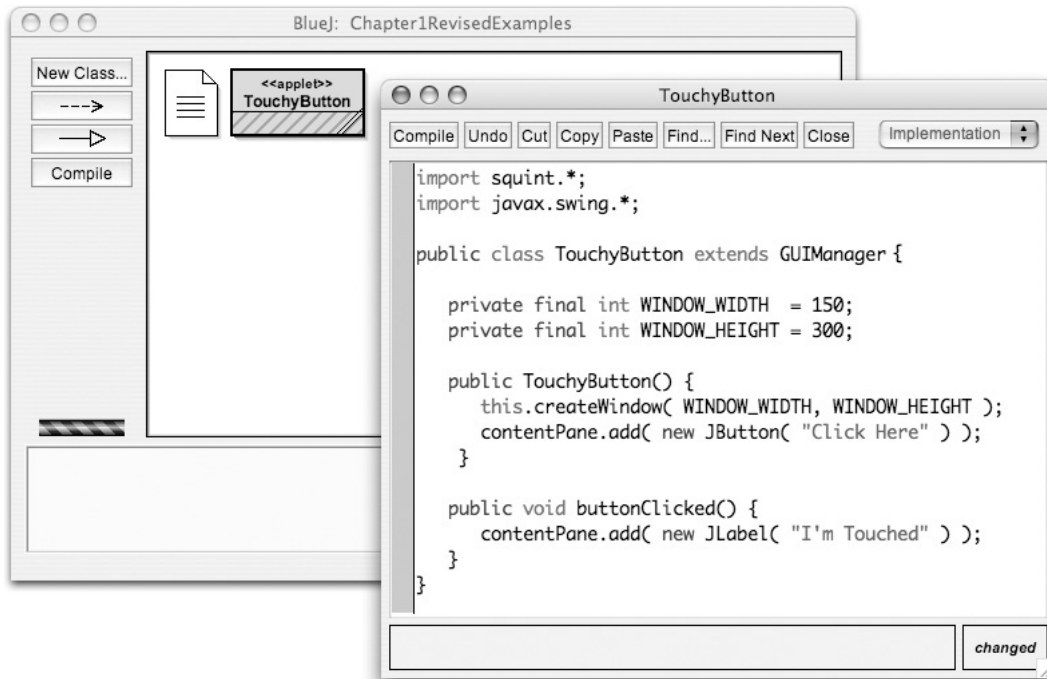


Figure 1.8: Entering the text of TouchyButton under BlueJ

this is done, we can ask Java to run the program by depressing the mouse on the icon that represents the TouchyButton class within the BlueJ project window and selecting the “new TouchyButton” item from the pop-up menu that appears. BlueJ will then display a new window controlled by the instructions included in our program. When the mouse is clicked on the button in this window, the words “I’m touched” will appear as shown in Figure 1.9.

With Eclipse, compiling a program and running it can be combined into a single step. We first create what Eclipse calls a *run configuration*. This involves specifying things like the size of the window created when the program starts running. We will not discuss the details of this process here. Once we have created a run configuration, we can compile and run our program by pressing an icon displayed at the top of the Eclipse window. Like BlueJ, Eclipse then displays a new window in which we can interact with our program.

In this discussion of how to enter and run a program we have overlooked one important fact. It is quite likely that you will make a mistake at some point in the process, leaving the IDE confused about how to proceed. As a result, in order to work effectively with an IDE you need some sense of what kinds of errors you are most likely to make and how the IDE will react to them. We will return to this issue after we teach you a bit more about how to actually write Java programs.

1.5 Graphical User Interface Components

Most current computer programs interact with their users through mechanisms like buttons, menus, and scroll bars. These mechanisms are called Graphical User Interface Components, or GUI com-

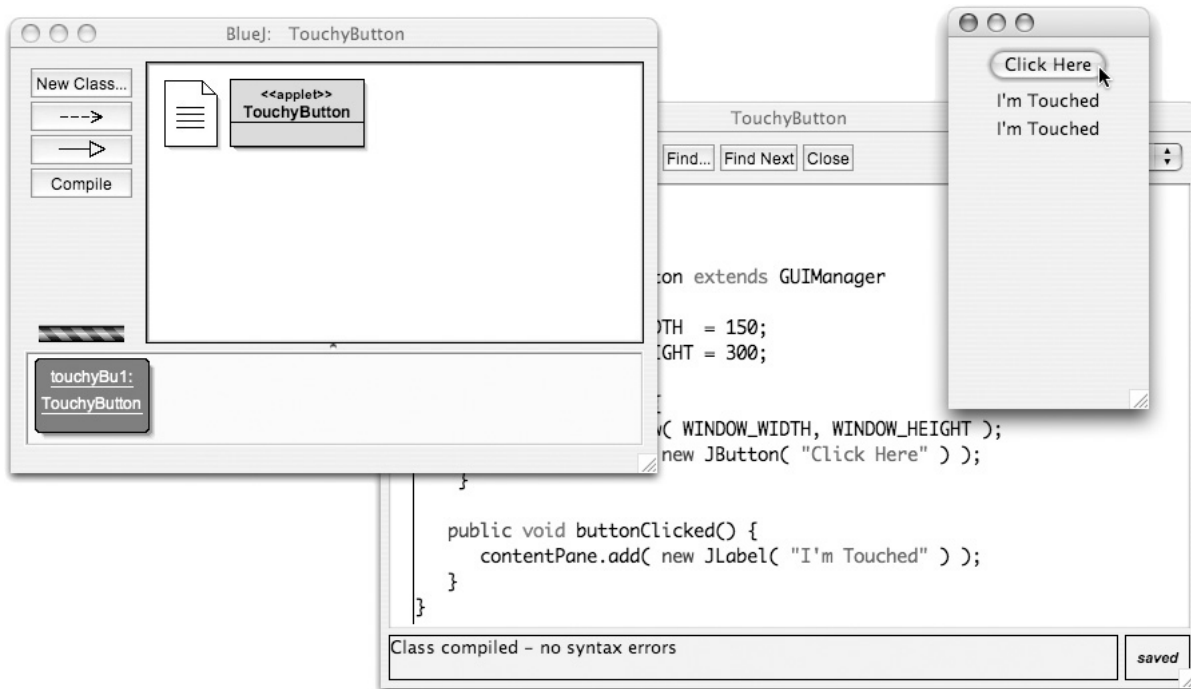


Figure 1.9: Running a program under BlueJ

ponents for short. The Java program we have considered thus far has involved one obvious example of a GUI component, a button. In fact, this program uses two forms of GUI components. The phrase displayed by the program, “I’m touched”, is clearly part of the means through which the program interacts with its users. As such, even though clicking on this phrase doesn’t have any effect, it is still part of the program’s graphical user interface. Such phrases are called *labels*.

In this section we will discuss the use of buttons and labels in more detail and we will introduce several additional types of GUI components that we will use in the Java programs we write. In particular, we will show you how to include fields where users can type text into your programs and how to create simple menus.

Your knowledge of Java is still so limited at this point that we won’t be able to do much with these GUI components. For example, although you will learn how to add a text field to a program’s interface, you won’t have any way to determine what the user has typed into that field. Introducing these components now, however, will enable us to explore several aspects of how GUI components are used in programs. In particular, we will learn more about how a program can be written to respond when users modify the settings of GUI components and we will learn a little about how to control how the GUI components that make up a program’s interface are arranged relative to one another in the program’s window.

1.5.1 Constructing GUI Components

Let’s begin by taking a closer look at some of the details of the `TouchyButton` program shown in Figure 1.4. If you examine the line in the constructor of `TouchyButton` that instructs the computer

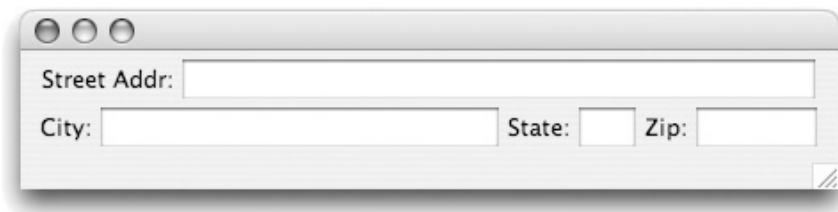


Figure 1.10: An interface for entering postal addresses

to display a button:

```
contentPane.add( new JButton( "Click Here" ) );
```

you should notice that it is very similar to the line from the `buttonClicked` method that tells the computer to display the phrase “I’m Touched”:

```
contentPane.add( new JLabel( "I’m Touched" ) );
```

Both lines are invocations instructing the `contentPane` to add additional components to the display. In addition, the phrases used to describe the arguments to these two method invocations have very similar structures. Both phrases have the form

```
new TypeOfObject( . . . )
```

In Java, a phrase of this form is called a *construction*. Each construction tells the computer to create a new object of the kind described by the “*TypeOfObject*” specified. We can use constructions like these to create all kinds of GUI components, and, as we will see later, many other objects within our programs.

The actual details of how GUI components should behave in a Java program are described as part of the Swing library that is imported for use by this program. The designers of Swing decided to name the class of objects that implement buttons `JButton` and to name pieces of text displayed alone on the screen `JLabels`. Therefore, we have to include these names to describe the types of objects we would like created by the two constructions in the `TouchyButton` program.

As in a method invocation, extra information required to construct the desired object can be provided in the parentheses after the specification of the type of the object to be constructed. Again, such pieces of information are called arguments or actual parameters. When creating a `JButton`, we provide the words we would like displayed in the button as an actual parameter. These words must be enclosed in quotes as shown.

Once you understand the basic form of a construction, it is easy to learn how to construct GUI components of other types. All you need to know is the name assigned by Swing to the type of component you want to create and the actual parameters to use to specify the details of the components. This information is provided in the documentation for the Swing libraries that is available online.

For example, suppose you were working on a program in which the user needed to enter a postal address and you therefore wanted to provide components in your program’s interface like those shown in Figure 1.10. First, you should recognize that there are only two types of GUI components used in this interface. The phrases “Street Addr:”, “City:”, “State:”, and “Zip:” are

```

public AddressWindow() {
    this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

    contentPane.add( new JLabel( "Street Addr:" ) );
    contentPane.add( new JTextField( 29 ) );

    contentPane.add( new JLabel( "City:" ) );
    contentPane.add( new JTextField( 18 ) );

    contentPane.add( new JLabel( "State:" ) );
    contentPane.add( new JTextField( 2 ) );

    contentPane.add( new JLabel( "Zip:" ) );
    contentPane.add( new JTextField( 5 ) );
}

```

Figure 1.11: Code to construct the interface shown in Figure 1.10

just examples of `JLabels`. The fields where the user would type the required information are examples of what Swing calls `JTextFields`.

Given that you know that the fields are called `JTextFields`, you now know that the constructions that instruct the computer to create them will all take the form

```
new JTextField( . . . )
```

All we have to figure out is what, if any, actual parameters should take the place of the “. . .”s.

If you look at Figure 1.10 again, you can see that the only difference between one of the `JTextFields` used in this example and another is how large each field is. The actual parameter in the construction is used to control the size of a `JTextField` in a fairly simple way. Each time you create a `JTextField`, you specify the number of characters you expect the user to type as an argument in the construction. For example, the construction for the zip code field would look like

```
new JTextField( 5 )
```

while the state field would be created by the construction

```
new JTextField( 2 )
```

This is all you need to understand the code used to construct the desired components shown in Figure 1.11.

There are other options available to you when you want to create a `JTextField`. Sometimes it is useful to be able to place some text in a `JTextField` when it is first created. Suppose that you first used a window like that shown in Figure 1.10 to have a user enter a home address and then wanted to display a similar window in which they could enter a work address. You might want the `JTextField` for the state information to appear already filled in with the state that was entered for the home address in the hope of saving the user a bit of time in the common case that they lived and worked in the same state. In a situation like this, you can provide two arguments in the

`TextField` constructor. For example, you could create a `TextField` that would initially contain the abbreviation for California by using the construction

```
new TextField( "CA", 2 );
```

The user of the program could still change the contents of the `TextField` when the window appeared if California was not the correct state for the work address.

We mention this to highlight a few important facts about constructions. First, while all of our preceding examples of constructions involved a single actual parameter, there are also examples of constructions that involve two or more parameters. There are also occasions when you will write constructions that involve no parameters. For example, the construction

```
new JButton()
```

would create an unlabeled button. Note that even if there are no arguments included in a construction, the parentheses are still required. This is also true for method invocations.

Second, we wanted you to see that in some cases Java provides you with several options to choose from when you want to construct a new object. Depending on the requirements of your program you will find some situations where you want `TextFields` that are initially empty and others where you want to fill them in. It is nice to have the option to do it either way.

Finally, we don't want you to think Java is more flexible than it really is. Having learned that you can write

```
new TextField( "CA", 2 );
```

to create a `TextField`, it might be tempting to assume that writing

```
new TextField( 2, "CA");
```

would have the same effect. This is not the case at all. If you included the second construction in a program, Java would reject the program as erroneous. Java is very particular about the order in which actual parameter values are included in constructions and method invocations. You have to check to make sure you are using the required order whenever you write a construction involving multiple actual parameters. We will tell you the correct parameter order for each method we introduce. Later, we will explain how to access the online documentation for the Swing library, which provides this information for all methods provided in the library.

There is one other GUI component we will find useful in many of our programming examples. In Swing, this component is called a `ComboBox` although it is more common to call it a menu or pop-up menu. Figure 1.12 includes two examples of `ComboBoxes` showing how they might look when they are idle and in use.

Obviously, the construction for one of these menus will look something like

```
new JComboBox( . . . )
```

The tricky part is specifying the argument for the construction. The argument can be used to specify what items should be included in the menu. Typically, we want several items in the menu, but Java expects us to provide this list of items wrapped up together as a single object. To do this, we have to use a special form of construction that packages up several pieces of text as a single entity. For example, to create the yes-no-maybe menus shown in Figure 1.12 we used constructions of the form



Figure 1.12: Examples of Java pop-up menus

```
new JComboBox( new String[] { "Yes", "No", "Maybe so" } )
```

The first `new` in this phrase tells the computer we want it to construct a new `JComboBox`. The second `new` tells it we want it to package up the list of words “Yes”, “No”, and “Maybe so” as a single object so that this object can be used as the argument that specifies the items that will be included in the menu. You may be wondering why the square brackets (“`[]`”) are included in this construction. Unfortunately, we won’t be able to give a satisfying explanation of the interpretation of this notation for a while. You will just have to accept it as another magic incantation until then. Fortunately, it is easy to imitate this example without fully understanding the notation. As an example, the construction

```
new JComboBox( new String[] { "AM", "PM" } )
```

could be used to create an AM/PM menu that might be helpful when entering the time of day.

1.5.2 Additional GUI Event-handling Methods

In the `TouchableButton` program, we defined a method with a special name, `buttonClicked`, as a way of telling the computer how we wanted our program to react when a user clicked on the button in our program. There are similar event-handling methods that can be defined to associate actions with other GUI components. In particular, instructions placed in a method named `textEntered` will be executed whenever a user presses the return or enter key while typing text in a `JTextField`, and instructions placed in a method named `menuItemSelected` will be executed when a user selects an item from a `JComboBox` menu.

As we mentioned above, at this point, you don’t know enough about Java to actually use the information a user types into a `JTextField` or to find out which item has been selected from a `JComboBox`. This makes it a bit hard to construct a realistic example to illustrate the use of these additional event-handling methods. We can, however, show you a somewhat silly example program to make things a bit more concrete. The program we have in mind will display menus like those shown in Figure 1.12. Whenever a person using this program selects an item from either of the menus shown, the program will encourage the user by enthusiastically displaying the message “What a good choice!” as shown in Figure 1.13. Just as our `TouchableButton` program would fill the window with copies of the message “I’m Touched” if its button was pressed several times, this program will display multiple copies of its message if the user selects menu items repeatedly. Note that at this point, when the `menuItemSelected` method is executed, not only doesn’t it know which menu item was selected, it can’t even tell whether the item was selected from the menu on the left or the menu on the right. This is an issue we will address in the next chapter. The complete code for the program is shown in Figure 1.14.

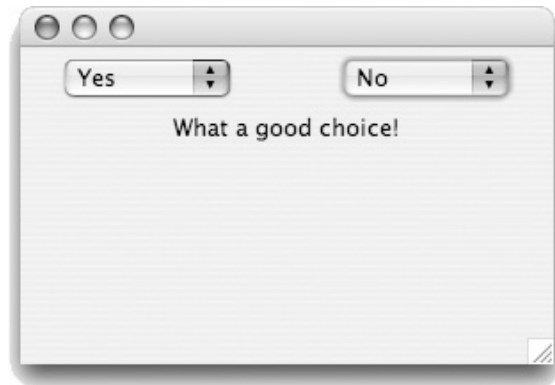


Figure 1.13: Reacting to JComboBox events

```
import squint.*;
import javax.swing.*;

public class TwoComboBoxes extends GUIManager {
    private final int WINDOW_WIDTH = 300;
    private final int WINDOW_HEIGHT = 200;

    public TwoComboBoxes() {
        this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );
        contentPane.add(
            new JComboBox( new String[] { "Yes", "No", "Maybe so" } )
        );
        contentPane.add(
            new JComboBox( new String[] { "Yes", "No", "Maybe so" } )
        );
    }

    public void menuItemSelected() {
        contentPane.add( new JLabel( "What a good choice!" ) );
    }
}
```

Figure 1.14: Handling JComboBox selection events

The code shown in Figure 1.14 illustrates one simple but useful fact about the process of entering your Java code. You don't have to type each command you write on a single line. Note that in typing in the command

```
contentPane.add(  
    new JComboBox( new String[] { "Yes", "No", "Maybe so" } )  
);
```

we split it up over three lines so that the construction of the `JComboBox` would stand out on a line by itself. Java allows you to start a new line anywhere other than in the middle of a word or in the middle of a text surrounded by quotes. That is, while the example above is fine, Java would not be happy if you typed

```
contentPane.add( new JCombo  
    Box( new String[] { "Yes", "No", "Maybe so" } )  
);
```

because the word `JComboBox` has been split into two words. Java would also complain if you entered the code

```
contentPane.add( new JComboBox( new String[] { "Yes", "No", "Maybe  
    so" } )  
);
```

because the quoted text "Maybe so" has been split over two lines. Basically, outside of quoted text, wherever you can type a single space in a Java program, you can type as many spaces, tabs, and/or new lines as you like. You can (and should!) use this feature to ensure that your code is formatted in a way that will make it as easy to read and understand as possible.

1.5.3 GUI Component Layout

When a program's interface involves a complex collection of buttons, menus, and other GUI components, the physical layout of these components on the screen can become an important concern. You may have noticed, however, that in all the example code we have shown you there have never been any instructions to the computer that explicitly say anything about how to arrange the components on the screen. The truth is that we are cheating a bit.

Java provides extensive mechanisms to enable a programmer to tell it how the components of a program's interface should be arranged. Unfortunately, learning how to use these mechanisms takes time, and there are many other things that are more important to learn as a beginning programmer. Accordingly, in our examples we have been using a very simple (but limited) approach to the layout of our GUI interfaces that we will expect you to use in many of your programs as well.

In the Java mechanisms for GUI layout, an object called a *layout managers* is associated with the content pane in which GUI components are displayed. The program can control the positioning of components by giving instructions to the layout manager. By default, a very simple layout manager is associated with the content pane of every program that is written by extending `GUIManager`. This layout manager is called a `FlowLayout`. With this layout manager, there are really only two factors that determine the arrangement of components in a program's window — the order in which the components are added to the window and the size of the window.

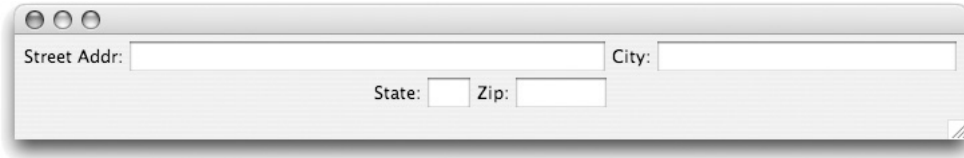


Figure 1.15: A component layout illustrating `FlowLayout` centering

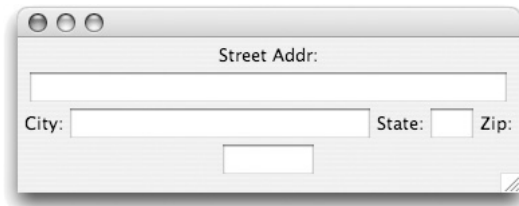


Figure 1.16: A component layout where `FlowLayout` falls short

The operation of a `FlowLayout` manager can be understood by thinking about how a word processor places words on a page as you type. Basically, as you type words into a word processor, it places as many words as it can on the first line. Then, when it reaches the point that it can't fit your next word on the first line, it starts a second line of words. Again, it keeps putting the words you enter on this second line until the line fills up, then it starts a third line and so on. A `FlowLayout` manager does the same thing with GUI components. It puts as many GUI components as it can fit on the first line of your screen, then it fills up a second line, and so on until you stop adding components. Take a quick look at the layout of the components shown in Figure 1.10. You can see that it was only possible to fit two components, the `JLabel` "Street Addr:" and one `JTextField`, on the first line. The remaining six components all fit on the second line. Importantly, notice that all eight components appear in the order in which they were added to the content pane.

The big difference between the way a `FlowLayout` positions components and a word processor positions words is that once a line is filled, the `FlowLayout` will center all the components on that line within the window rather than always starting at the left edge of the window. To see this, suppose that we increased the value associated with `WINDOW_WIDTH` in the program that creates the components shown in Figure 1.10 so that the program window becomes wider. The resulting layout might look like that shown in Figure 1.15. In this wider window, the first four GUI components added to the content pane now fit across the first line of the window. This leaves only four components remaining. These components don't fill the second line of the window, so the `FlowLayout` manager centers them in the window.

The display of the components in Figure 1.15 isn't quite as nice as that in Figure 1.10, but it is still reasonable. That is where we are cheating. The simple process followed by a `FlowLayout` manager does not always result in reasonable positioning of the components. For example, if we change the window size again to make it narrower but taller, we can get a layout like that shown in Figure 1.16. It would be much nicer to move the `JLabel` "Zip:" to the next line so that it would still be attached to the `JTextField` it is supposed to label, but the `FlowLayout` manager does not know enough to do this.

Despite such limitations, we believe that for our purposes, the simplicity of the `FlowLayout` manager outweighs its weaknesses. We will introduce one mechanism in the next chapter that can be used to fix problems like the misplaced “Zip” shown in Figure 1.16, but beyond that we will postpone exploring the more advanced features Java provides to control component layout. There are many more interesting aspects of programming in Java that we would rather explore first.

1.6 To Err is Human

We all make mistakes. Worse yet, we often make the same mistakes over and over again.

If we make a mistake while giving instructions to another person, the other person can frequently figure out what we really meant. For example, if you give someone driving directions and say “turn left” somewhere you should have said “turn right” chances are that they will realize after driving in the wrong direction for a while that they are not seeing any of the landmarks the remaining instructions mention, go back to the last turn before things stopped making sense, and try turning in the other direction. Our ability to deal with such incorrect instructions, however, depends on our ability to understand the intent of the instructions we follow. Computers, unfortunately, never really understand the instructions they are given so they are far less capable of dealing with errors.

There are several distinct types of errors you can make while writing or entering a program. The computer will respond differently depending on the type of mistake you make. The first type of mistake is called a *syntax error*. The defining feature of a syntax error is that the IDE can detect that there is a problem before you try to run your program. As we have explained, a computer program must be written in a specially designed language that the computer can interpret or at least translate into a language which it can interpret. Computer languages have rules of grammar just like human languages. If you violate these rules, either because you misunderstand them or simply because you make a typing mistake, the computer will recognize that something is wrong and tell you that it needs to be corrected.

The mechanisms used to inform the programmer of syntactic errors vary from one IDE to another. Eclipse constantly examines the text you have entered and indicates fragments of your program that it has identified as errors by underlining them and/or displaying an error icon on the offending line at the left margin. If you point the mouse at the underlined text or the error icon, Eclipse will display a message explaining the nature of the problem. For example, if we accidentally left out the curly brace that terminates the body of the `TouchableButton` constructor while entering the program shown in Figure 1.4, Eclipse would underline the semicolon at the end of the last line of the constructor. Pointing the mouse at the underlined semicolon would cause Eclipse to display the message “Syntax error, insert ‘}’ to complete MethodBody” as shown in Figure 1.17.

The bad news is that your IDE will not always provide you with a message that pinpoints your mistake so clearly. When you make a mistake, your IDE is forced to try to guess what you meant to type. As we have emphasized earlier, your computer really cannot be expected to understand what your program is intended to do or say. Given its limited understanding, it will often mistake your intention and display error messages that reveal its confusion. For example, if you type

```
public void buttonClicked {
```

instead of

```
public void buttonClicked() {
```

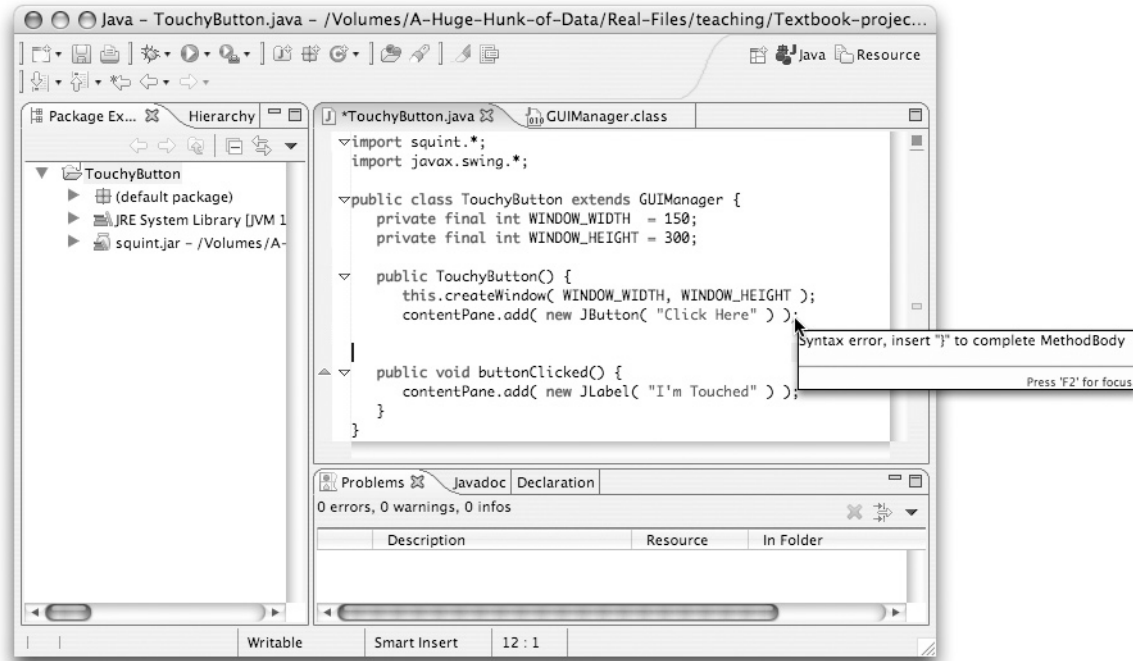


Figure 1.17: Eclipse displaying a syntax error message

as the header of the `buttonClicked` method of our example program, Eclipse will underline the word “`buttonClicked`” and display the message “`void is an invalid type for the variable buttonClicked.`” In such cases, the error message is more likely to be a hindrance than a help. You will just have to examine what you typed before and after the position where the IDE identified the error and use your knowledge of the Java language to identify your mistake.

BlueJ is more patient about syntax errors. It ignores them until you press the Compile button. Then, before attempting to compile your code it checks it for syntactic errors. If an error is found, BlueJ highlights the line containing the error and displays a message explaining the problem at the bottom of the window containing the program’s text. Figure 1.18 shows how BlueJ would react to the mistake of leaving out the closing brace at the end of the `TouchyButton` constructor. Note that different IDEs may display different messages and, particularly in the case where the error is an omission, associate different points in the code with the error. In this case, Eclipse flags the line before the missing brace while BlueJ highlights the line after the error.

A program that is free from syntax errors is not necessarily a correct program. Think back to the instructions for performing calculations that were designed to leave you thinking about Danish elephants that we presented in Section 1.1. If while typing these instructions we completely omitted a line, the instructions would still be grammatically correct. Following them, however, would no longer lead you to think of Danish elephants. The same is true for the example of saying “left” when you meant to say “right” while giving driving directions. Mistakes like these are not syntactic errors; they are instead called *logic errors* or simply *bugs*. They result in an algorithm that doesn’t achieve the result that its author intended. Unfortunately, to realize such a mistake has been made, you often have to understand the intended purpose of the algorithm. This is exactly what

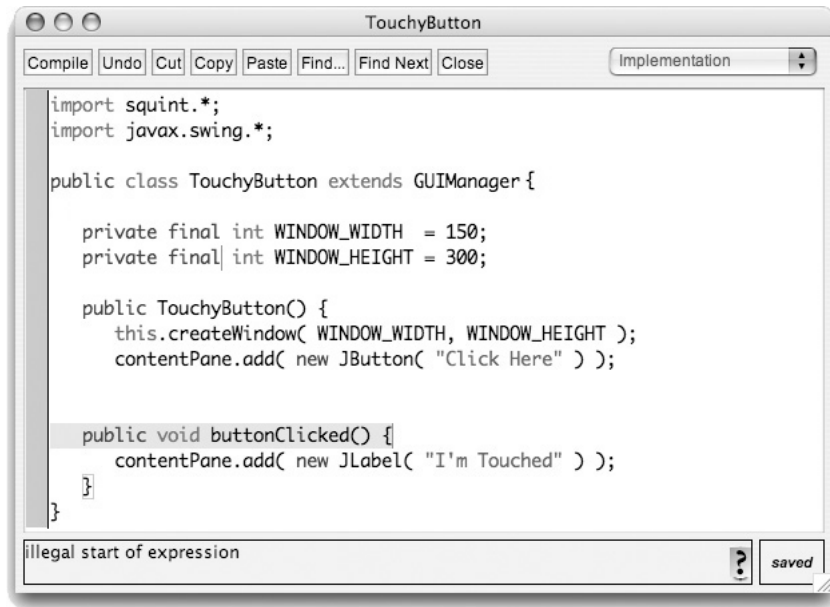


Figure 1.18: BlueJ displaying a syntax error message

computers don't understand. As a result, your IDE will give you relatively little help correcting logic errors.

As a simple example of a logical error, suppose that while typing the `buttonClicked` method for the `TouchyButton` program you got confused and typed `menuItemSelected` instead of `buttonClicked`. The result would still be a perfectly legitimate program. It just wouldn't be the program you meant to write. Your IDE would not identify your mistake as a syntax error. Instead, when you ran the program it just would not do what you expected. When you clicked the button, nothing would happen at all. Since there is no `JComboBox` in which you could select an item, the code in the `menuItemSelected` method would never be executed.

This may appear to be an unlikely error, but there is a very common error which is quite similar to it. Suppose instead of typing the name `buttonClicked` you typed the name `buttonlicked`. Look carefully. These names are not the same. `buttonlicked` is not the name of one of the special event handling methods discussed in the preceding sections. In more advanced programs, however, we will learn that it is sometimes useful to define additional methods that do things other than handle events. The programmer is free to choose any name for such a method. `buttonlicked` would be a legitimate (if strange) name for such a method. That is, a program containing a method with this name has to be treated as a syntactically valid program by any Java IDE. As a result, your IDE would not recognize this as a typing mistake, but when you ran the program Java would think you had decided not to associate any instructions with button clicking events. As before, the text "I'm Touched" would never appear in the window.

There is an even more subtle way this error can occur. Suppose that instead of typing `buttonClicked` you typed `buttonlicked`. That is, you didn't capitalize the "c" in "clicked". Java considers upper and lower case letters used in names completely distinct. Therefore, `buttonlicked` is considered just as different from `buttonClicked` as the name `buttonlicked`. Changing the capi-

tal “C” to a lower case “c” would therefore confuse Java enough that the code in the method would not be executed when a button was clicked.

There are many other examples of logical errors a programmer can make. In larger programs the possibilities for making such errors just increase. You will find that careful, patient, thoughtful examination of your code as you write it and after errors are detected is essential.

1.7 Summary

Programming a computer to say “I’m Touched.” is obviously a rather modest achievement. In the process of discussing this simple program, however, we have explored many of the principles and practices that will be developed throughout this book. We have learned that computer programs are just collections of instructions. These instructions, however, are special in that they can be followed mechanically, without understanding their actual purpose. This is the notion of an algorithm, a set of instructions that can be followed to accomplish a task without understanding the task itself. We have seen that programs are produced by devising algorithms and then expressing them in a language which a computer can interpret. We have learned the rudiments of the language we will explore further throughout this text, Java. We have also explored some of the tools used to enter computer programs, translate them into a form the machine can follow, and run them.

Despite our best efforts to explain how these tools interact, there is nothing that can take the place of actually writing, entering and running a program. We strongly urge you to do so before proceeding to read the next chapter. Throughout the process of learning to program you will discover that it is a skill that is best learned by practice. Now is a good time to start.