Chapter 5

Pro-Choice

In order to behave in interesting ways, programs need to be able to make choices. To construct such programs, we need a way to write commands that are conditioned on user input and events that have already occurred. For example, instead of only being able to say "send the message", we need to learn how to say "if the user entered a valid password, send the message." In this chapter we will present a new form of instruction called the `if` statement. This type of statement is designed precisely to enable us to express choices in Java.

Learning about `if` statements will enable you to write much more complex programs. As a result, it is important not just to learn the grammatical structure of this new language mechanism, but also to develop an understanding of how to use it to construct programs that are clear and easy to understand. With this in mind, we both present the basic syntax associated with the Java `if` statement and explore some common patterns that can be employed when using `if` statements in programs.

5.1 Either/or Questions

To illustrate the Java `if` statement, we will explore the implementation of a calculator program based on the numeric keypad interface we introduced in Section 3.4. Rather than plunging into an attempt to implement a full-featured calculator, we will start by writing a very simple calculator program. Figure 5.1 shows the interface for the program we have in mind and illustrates how it will respond as a user clicks on some of the buttons in its interface.

The program displays ten buttons labeled with digits and one button labeled "Add to total". It also contains two text fields. Each time a digit button is pressed, its label will be added to the end of the sequence of digits in the first text field. When the "Add to total" button is pressed, the numerical value of the digits in the first text field will be added to the value displayed in the second text field, the result will appear in the second text field, and the first text field will be cleared. Obviously, it might be more accurate to describe this program as an "adding machine" than as a calculator.

In Figure 5.1 we show how the program would behave if a user pressed the sequence of buttons "4", "2", "Add to total", "8", and then "Add to total" again. The image in the upper left corner of the figure shows how the program should look when it first begins to execute. After button "4" was pressed, the digit "4" would be added to the first text field as shown in the image at the upper right in the figure. The arrow labeled "2" leads to a picture of how the window would look after the

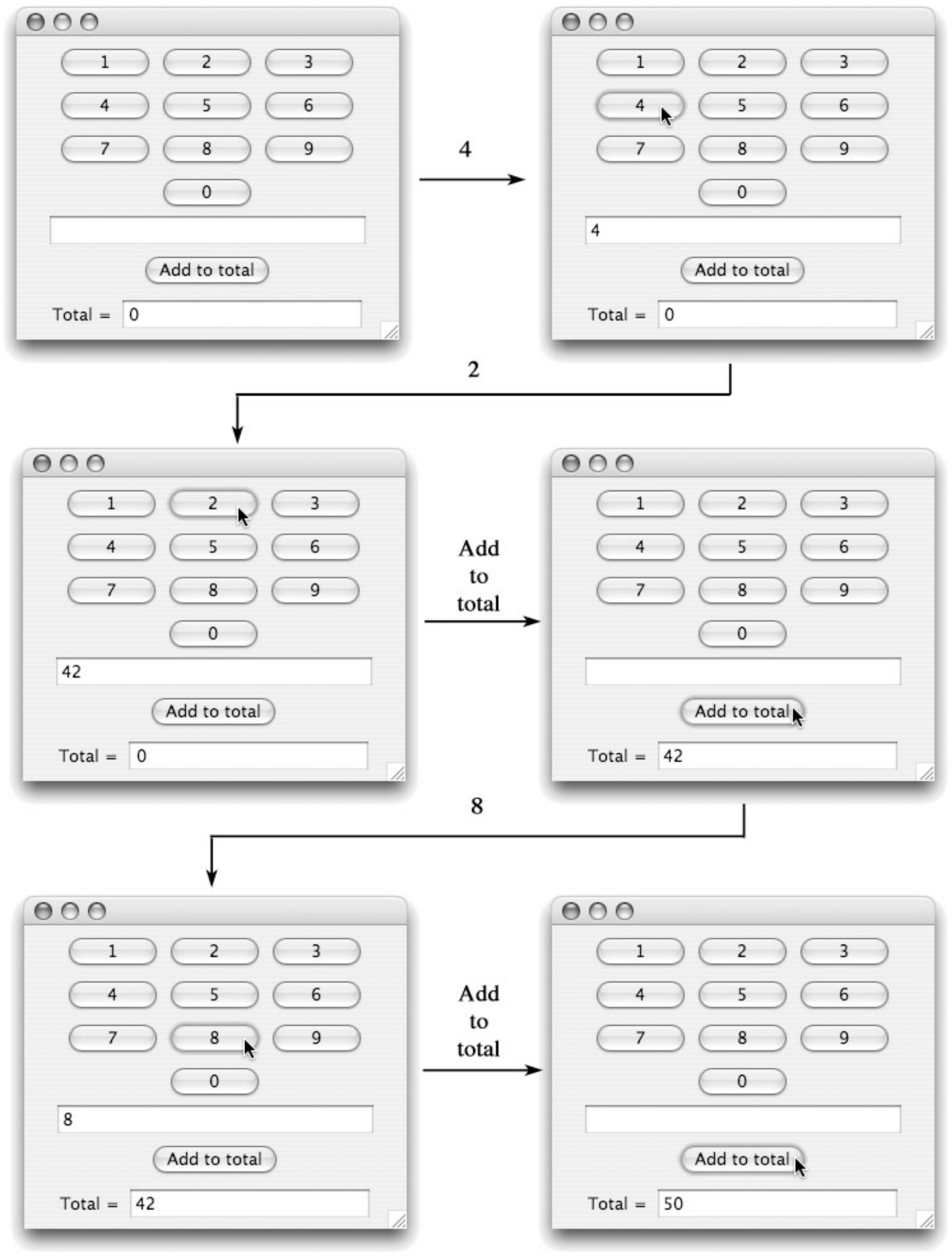


Figure 5.1: Interface for an “adding machine” program

```
// A program that acts like a calculator that only does additions
public class AddingMachine extends GUIManager {
   // Change these values to adjust the size of the program's window
   private final int WINDOW_WIDTH  = 300, WINDOW_HEIGHT = 260;

   // Dimensions for fields to display entry and total
   private final int ENTRY_WIDTH = 20;
   private final int DISPLAY_WIDTH = 15;

   // Used to display sequence of digits selected
   private JTextField entry;

   // Used to display current total
   private JTextField totalDisplay;

   // Used to request that the total be updated
   private JButton addButton;

   // Keeps track of the current sum
   private int total = 0;
```

Figure 5.2: Instance variables for an adding machine

next button, "2", had been pressed. The number "42" would be displayed in the first text field. At that point, we assume the user pressed "Add to total". Since nothing had previously been added to the total, its value would be 0. Adding 42 to 0 produces 42, which therefore would appear in the "Total =" text field at the bottom of the program window. At the same time, the upper text field would be cleared. Therefore, when the next button, "8" was pressed, the digit 8 would appear alone in the upper text field. Finally, pressing "Add to total" again would cause the program to add 8 to 42 and display the result, 50, in the total field.

The code required to create this program's interface is shown in Figures 5.2 and 5.3. The first figure shows the instance variables declared for this program, and Figure 5.3 shows the declaration of the constructor for the class. This code is quite similar to the code we showed earlier in Figure 3.12 while discussing how to implement a simple numeric keypad, but there are two significant differences.

For this program, we need to add a button labeled "Add to total". Unlike the buttons that display the digits, we will need to have a variable name associated with this special button. Accordingly, we include a declaration for the needed variable, `addButton`, before the constructor and initialize this variable's to refer to a new button within the constructor.

We also need to keep track of the total and display it on the screen. This requires two variables. First, we define an `int` variable named `total` to hold the current value of the sum of all numbers that have been entered. Second, we declare and initialize a variable, `totalDisplay` that refers to a `JTextField` used to display the total. The construction used to create `totalDisplay` includes the parameter value `"0"` so that this field will display the initial value associated with `total` when the program begins execution.

```
// Create and place the keypad buttons in the window
public AddingMachine() {
   this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

   contentPane.add( new JButton( "1" ) );
   contentPane.add( new JButton( "2" ) );
   contentPane.add( new JButton( "3" ) );
   contentPane.add( new JButton( "4" ) );
   contentPane.add( new JButton( "5" ) );
   contentPane.add( new JButton( "6" ) );
   contentPane.add( new JButton( "7" ) );
   contentPane.add( new JButton( "8" ) );
   contentPane.add( new JButton( "9" ) );
   contentPane.add( new JButton( "0" ) );

   entry = new JTextField( ENTRY_WIDTH );
   entry.setEditable( false );  // Prevent the user from typing in the field
   contentPane.add( entry );
   addButton = new JButton( "Add to total" );
   contentPane.add( addButton );

   JPanel totalPane = new JPanel();
   totalPane.add( new JLabel( "Total = " ) );
   totalDisplay = new JTextField( total, DISPLAY_WIDTH );
   totalPane.add( totalDisplay );
   contentPane.add( totalPane );
}
```

Figure 5.3: Constructor for an adding machine

5.1.1 Numbers and `Strings`

Unlike other programs we have considered, the action performed when a button is clicked in this program will depend on which button is pressed. If any of the digit keys is pressed, the corresponding digit should simply be added to the entry field. If the "Add to total" button is pressed, on the other hand, the contents of the entry field should be added to the contents of the total field. In the next section, we will see how to use an `if` statement to specify when each of these actions should be performed. Before considering how we can use an `if` statement, we should make sure we know exactly how to write code that will perform the desired actions individually.

Both of the actions that might be performed when a button is clicked involve working with numbers. When writing Java programs that manipulate numbers, it is important to understand that there are several distinct ways to represent numbers. The programmer must select the approach to representing numbers that is most appropriate for the task at hand.

We have, in fact, already introduced two of the ways that numbers can be represented in a Java program. First, we have used the type `int` to represent numeric data. In the preceding section, we suggested that we would use an `int` variable to represent the total of all the numbers added together using our simple calculator. The other way we have seen that numbers can be represented is as `Strings`. Anything that is typed in quotes or associated with a `String` variable name in Java is treated as a `String` by the language, even if it also looks like a number. Thus, while it is clear that the value represented by

```
"Click me"
```

is a `String`, Java also sees the value

```
"42"
```

as a `String` rather than as a number because the digits are surrounded by quotes.

The distinction within the Java language between a number represented as an `int` and a number represented as a `String` provides a means by which you can tell Java exactly how you want certain operations to be performed on a number. Consider again the example used in the preceding section. In that example, we assumed a user pressed the buttons "4", "2", "Add to total", "8", and finally "Add to total" in sequence. Figure 5.1 showed the expected response to each of these actions. We indicated that the last user action, pressing the "Add to total" button, should cause the program to "add" 8 to 42 yielding 50.

Suppose, that the user had not pressed the "Add to total" button between pressing the "2" key and the "8" key. In that case, pressing the "8" key would be interpreted as specifying a third digit for the number being entered. Therefore, we would expect the program to "add" the digit 8 to the digits already entered, 4 and 2, and display the number 428 in the text field above the "Add to total" button.

From this example, we can see that there are two different interpretations we might associate with the word "add". When we see the digits as parts of a larger number, then we think of numerical addition when we say "add". When we think of the digits as individual symbols, then when we say "add" we mean that a digit should be appended or concatenated to an existing collection of digits.

In Java, the plus sign operator is used to describe both of these possible interpretations of "add". When a program applies the plus sign to operands represented as `int` values, Java interprets the plus sign as numeric addition. On the other hand, if either of the operands of a plus sign is a `String`, Java interprets the operator as concatenation, even if all of the symbols in the `String` are digits. Accordingly, to ensure that Java applies the correct interpretation to the `+` operator in our calculator program, we must ensure that the operands to the operator are `int` values when we want addition and `String` values when we want concatenation.

To do this, we have to understand how Java decides whether a particular value in our program should be treated as a `String` or as an `int`. Recall that values in our programs are described by expressions and that we have identified five types of expressions: literals, variables, accessor method invocations, constructions and formulas. Java has simple rules for determining the type of data described by each of these forms of expressions.

When we use a literal to describe a value in a program, Java treats the value as a `String` if we surround it by quotes even if all of the symbols between the quotes are digits. On the other hand, if a literal is a sequence of digits not surrounded by quotes, then Java treats the value as an `int`.

When we use a variable as an expression, Java depends on the type name the programmer used in the variable's declaration to determine how to interpret the value associated with the variable name. If we declare

```
String entry;
int total;
```

then Java assumes that the expression "`entry`" describes a `String` and that the expression "`total`" describes an `int`.

The type of an invocation depends on the method used in the invocation. When you write programs in which you define your own accessor methods, you will have to provide the name of the type of value an invocation will produce in the definition of the method. For methods provided as part of existing libraries, the type of value produced when the method is invoked is provided in the documentation of the library. For example, the `getText` method associated with `JTextField`s, `JButton`s, and `JLabel`s is known to produce a `String` value as its result, while the `getCaretPosition` method associated with `JTextField`s and `JTextArea`s produces an `int` result.

A construction of the form

```
new SomeName( . . . )
```

produces a value of the type whose name follows the word `new`.

Determining the type of value described by a formula can be a bit trickier because the type produced often depends on the types of the operands used. As explained above, if we write an expression of the form

```
a + b
```

Java may either interpret the plus sign as representing numeric addition or concatenation, depending on the types it associates with `a` and `b`. If Java's rules for associating types with expressions lead to the conclusion that both `a` and `b` produce `int` values, then Java will assume that `a + b` also produces an `int` value. On the other hand, if either `a` or `b` describes a `String` value, then Java will interpret the plus sign as concatenation and also conclude that `a + b` describes a `String`.

Within our adding machine program, we want Java to apply the concatenation operation when the user presses any of the buttons for the digits 0 through 9. This is easy to do. In the code for our simple keypad program shown in Figure 3.12 we used the instruction

```
        entry.setText( entry.getText() + clickedButton.getText() );
```

to add digits to the display as numeric keys were pressed. Because both of the operands to the plus sign in this statement are produced by `getText`, they will both be `String`s. Java will therefore associate the plus sign with concatenation. As a result, the same code can be used when any of the digit keys are pressed in our adding machine program.

The instructions we should use when the “Add to total” button is pressed require a bit more thought. We want code that will add the contents of the `entry` text field to our total using numeric addition. The obvious code to write would be

```
total = entry.getText() + total;
```

Unfortunately, Java will consider this statement illegal. The statement says to associate the name `total` with the result produced by the expression to the right of the equal sign. Because the `getText` method returns a string, Java will interpret the plus sign in this statement as a request to perform concatenation rather than addition. Concatenation produces a `String` value. Since we plan to declare `total` as an `int` variable, Java will conclude that this name cannot be associated with the `String` produced by concatenation and reject the statement.

In a situation like this, we have to help Java by adding phrases to our code that explicitly instruct it to attempt to convert the digits in the string produced by `entry.getText()` into an `int` value. For such situations, Java includes a method named `Integer.parseInt`. `Integer.parseInt` takes a `String` as a parameter and returns its value when interpreted as an `int`.[1] Thus, assuming that the `entry` field of our program contains the string `"8"`, then

```
Integer.parseInt( entry.getText() )
```

would produce the integer value 8, and

```
42 + Integer.parseInt( entry.getText() )
```

would produce 50. Therefore,

```
total = total + Integer.parseInt( entry.getText() );
```

is a statement we can use to tell Java to add the value of the digits in the text field to our running total.

There is one last representation issue we need to deal with while completing the body of this program’s `buttonClicked` method. After we have updated the total, we need to update the contents of the `totalDisplay` text field to display the new result. We cannot simply say

```
totalDisplay.setText( total );
```

because just as `getText` produces a `String` value, the `setText` method expects to be provided a `String` to display rather than an `int`. Again, we have to do something more explicit to tell Java to convert between one representation and another. We do not, however, need a special method like `Integer.parseInt` to convert an `int` to a `String`. Instead, we can take advantage of the way Java interprets the plus sign. Consider the expression

[1]When `Integer.parseInt` is used, the argument provided must be a string that contains only digits. If its argument contains letters or anything else other than digits, an error will result as your program runs. To make sure that this will not happen to our program, we have used `setEditable` to ensure that the user cannot type arbitrary data into the field.

```
// Add digits to the display or add the display to the total
public void buttonClicked( JButton clickedButton ) {

     if ( clickedButton == addButton ) {
         total = total + Integer.parseInt( entry.getText() );
         totalDisplay.setText( "" + total );
         entry.setText( "" );

     } else {
         entry.setText( entry.getText() + clickedButton.getText() );

     }
 }
```

Figure 5.4: `buttonClicked` method for an adding machine

```
"" + total
```

The first operand in this formula a very special `String`, the `String` containing no symbols at all. It is usually called the *empty string*. Even though it is empty, this `String` will cause Java to interpret the plus sign in the expression as a request to perform concatenation. Of course, concatenating the digits that represent the value of `total` onto a `String` that contains nothing will produce a `String` containing just the digits that represent the value of `total`. Accordingly, we can use the statement

```
totalDisplay.setText( "" + total );
```

to update the display correctly.

5.1.2 A Simple `if` Statement

Now that we know the correct instructions for the actions that our adding machine might perform when a button is clicked, we can learn how to use an `if` statement to specify how the computer should determine which set of instructions to follow.

This code will all be placed in the definition of our program's `buttonClicked` method. To choose the right code to execute, we will have to know which button was clicked. Therefore, the header for the `buttonClicked` method must tell the system that we want to associate a name with the button that was clicked. We will use the method header

```
public void buttonClicked( JButton clickedButton ) {
```

so that we can use the name `clickedButton` to refer to the button that was clicked.

When the computer starts to execute the code in `buttonClicked` there are two cases of interest. Either the "Add to total" button was clicked, in which case the formal parameter name `clickedButton` will refer to the same button as the instance variable `addButton`, or one of the digit buttons was pressed, in which case `clickedButton` and `addButton` will refer to different buttons. Based on this observation, we can tell the computer to decide which instructions should be executed by writing the `if` statement shown in the definition of `buttonClicked` found in Figure 5.4

The `if` statement is a construct that lets us combine simpler commands to form a larger, conditional command. The following template shows the general structure of the type of `if` statement we are using in Figure 5.4:

```
if ( condition ) {
      sequence of statements
} else {
      sequence of statements
}
```

The statement starts with the word `if` followed by what is called a *condition*. We will give a precise definition of a condition later, but basically a condition is a question with a yes or no answer. In our `buttonClicked` method, we use the condition

```
clickedButton == addButton
```

The double equal sign in Java indicates a test for equality. Thus, this condition asks if the two names refer to the same thing.

After the condition, we place two sequences of statements each surrounded by curly braces and separated by the word `else`. If the answer to the condition is "yes", then the computer will follow the instructions included in the first sequence of statements. Thus, the `if` statement in our `buttonClicked` method instructs the computer to update the total if `clickedButton` and `addButton` refer to the same button. If the answer to the condition in an `if` statement is "no", then the computer follows the instructions in the second sequence of statements. In our example, this is a sequence of just one instruction, the instruction that adds a single digit to the entry display.

5.2 `if` Statements are Statements

Consider the sentence

if it is raining, we will take the car.

According to the rules of English, this is indeed a sentence. At the same time, two of its parts

it is raining

and

we will take the car

would also be considered sentences if they appeared alone. As a result, the original sentence is classified as a compound sentence.

The Java `if` statement exhibits similar grammatical properties. An `if` statement is considered to be a form of statement itself, but it always contains subparts that would be considered statements if they appeared alone. Because it is composed of smaller components that are themselves statements, the `if` statement is an example of a *compound statement*.

Anywhere that we have said that Java expects you to provide a statement, you can use any kind of statement the language recognizes. In particular, the statements that form the subparts of an `if` statement do not have to be simple statement. They can instead be compound statements.

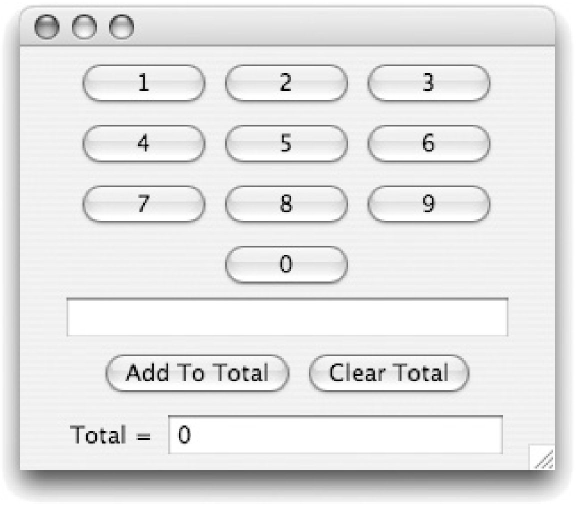


Figure 5.5: Interface for an adding machine with a “Clear total” button

This means that it is possible to include `if` statements among the sequences of statements that form the parts of a larger `if` statement. Such constructs are called *nested if statements*. Even though the basic form of an `if` statement offers only two possibilities, by nesting them it is possible to express choices involving more options.

To explore this ability, let’s add just one option to our adding machine program. As described in the preceding sections, our program is capable of adding up one sequence of numbers. Suppose that we had more than one sequence that we needed to total. At this point, the only way we could do this with our program would be to quit the program after computing the first total and then start it up again to add up the second sequence of numbers. Let’s fix this by adding a button to reset the total to zero.

A picture of what this improved program might look like is shown in Figure 5.5. To construct this interface, we need to add just one `JButton` to our window. We will assume that as part of this addition, we include the instance variable declaration

```
private JButton clearButton;
```

and an assignment of the form

```
clearButton = new JButton( "Clear Total" );
```

so that the name `clearButton` can be used to refer to the new button in our code.

As we did in the previous example, we should first determine the instructions that should be executed in each of the cases the program must handle. We already discussed the code needed for the cases in which the “Add to total” button or one of the numeric buttons is pressed. All that remains is the case where the “Clear total” button is clicked. In this situation, we want to set the total to 0 and clear the field used to display the total. This can be done using the instructions

```
total = 0;
totalDisplay.setText( "0" );
```

```
if ( clickedButton == addButton ) {
    total = total + Integer.parseInt( entry.getText() );
    totalDisplay.setText( "" + total );
    entry.setText( "" );
} else {
    . . .   // Need to find code to handle all the other buttons
}
```

Figure 5.6: Skeleton of `if` statement to support "Add to total" and "Clear total" buttons

Now, let us consider how to distinguish the three options that are possible with the addition of a "Clear total" button. A good place to start is by noticing that the `if` statement we used in Figure 5.4 would do one part of the job correctly. It selects the right code for the case when the button clicked is the "Add to total" button. Unfortunately, it does not always select the right code when a different button is used. Based on this observation, consider the skeletal code that is left if we keep the parts of that `if` statement that seem to correctly describe what we want in this situation, and temporarily replace the code that isn't quite appropriate for this new task with "..." as shown in Figure 5.6

Obviously, we still need to decide what statement(s) to use in the `else` branch where we have currently written "..." If the computer decides it should execute the statements in the `else` branch, that means it has already checked and determined that the button clicked was not the "Add to total" button. That is, if the computer gets to the `else` part of this `if` statement, there are no longer three possibilities to worry about! There are only two possibilities remaining. Either the user clicked on the "Clear total" button or the user clicked on a numeric button. We already know how to deal with 2-way choices like this. We can use an `if` statement of the form

```
if ( clickedButton == clearButton ) {
    total = 0;
    totalDisplay.setText( "0" );
} else {
    entry.setText( entry.getText() + clickedButton.getText() );
}
```

The critical observations are a) that the `if` statement shown above is itself a statement, and b) that this `if` statement correctly handles the two cases remaining when we know that the "Add to total" button was not clicked. Therefore, we can insert it where we had placed the comment saying we needed "to find code to handle all the other buttons" to produce the construct shown in Figure 5.7.

The construct in Figure 5.7 is an example of a *nested if statement*. When the computer needs to execute this code, it firsts checks to see if `clickedButton` and `addButton` refer to the same button. If they do, the group of statement that use `Integer.parseInt` to add the digits in the `entry` field to the total are executed. If `clickedButton` and `addButton` refer to different buttons, then the computer will begin to execute the statements after the word `else`. In this case, this means it will begin to execute another `if` statement. The computer begins the execution of this nested `if` statement by checking to see if `clickedButton` and `clearButton` refer to the same button. If they do, then the statements that set `total` equal to 0 will be executed. Otherwise, the execution of the `if` statement will complete with the statement that adds the digit on the button clicked to `entry`.

```
if ( clickedButton == addButton ) {
    total = total + Integer.parseInt( entry.getText() );
    totalDisplay.setText( "" + total );
    entry.setText( "" );
} else {
    if ( clickedButton == clearButton ) {
        total = 0;
        totalDisplay.setText( "" );
    } else {
        entry.setText( entry.getText() + clickedButton.getText() );
    }
}
```

Figure 5.7: Using a nested `if` statement to distinguish three cases

The ability to nest statements is not limited to a single level of nesting. The `if` statement shown in Figure 5.7 could itself be nested within a larger if statement. Just as the nesting of `if` statements in this example was used to handle a 3-way choice, deeper nesting can be used to handle even larger multi-way choices.

5.3 The Truth about Curly Braces

In this section, we provide a more precise explanation of how curly braces should be used when writing `if` statements. While the explanation and examples we have provided thus far accurately describe how curly braces are commonly used in Java `if` statements, they don't describe the exact rules of Java's grammar that determine correct usage.

We have indicated that the general form of an `if` statement in Java is

```
if ( condition ) {
    sequence of statements
} else {
    sequence of statements
}
```

This is not quite accurate. The actual rules of Java's grammar specify that the form of an `if` statement is

```
if ( condition )
    statement
else
    statement
```

That is, there are no curly braces and there are exactly two statements that are considered part of the `if` statement, the ones that appear before and after the word `else`. Following this rule the statement

```
if ( clickedButton == clearButton )
    total = 0;
else
    entry.setText( entry.getText() + clickedButton.getText() );
```

is a valid `if` statement, but all of the other examples of `if` statements we have provided in this chapter appear to be invalid! All our examples have had curly braces and many have had multiple statements between the condition and the `else`. How can these examples be legal if the actual rules for the `if` statement don't include these features?

The resolution of this apparent inconsistency rests on understanding exactly what the word "statement" means. In some situations, Java is willing to consider a construct composed of several statements as a single, albeit more complicated statement in its own right. As an example, the construct just shown above

```
if ( clickedButton == clearButton )
    total = 0;
else
    entry.setText( entry.getText() + clickedButton.getText() );
```

is a single Java statement even though it contains two lines that are themselves Java statements.

In an `if` statement, the fact that several independent statements are combined to form one logical statement is a secondary effect. Java, however, also provides a construct whose primary purpose is to group a sequence of statements into a unit that is logically a single statement. This construct is called the *compound statement* or *block*.

If we place curly braces around any sequence of statements and local variable declarations, Java considers the bracketed sequence of statements as a single statement. For example, while

```
total = 0;
totalDisplay.setText( "0" );
```

is a sequence of two statement, the construct

```
{
    total = 0;
    totalDisplay.setText( "0" );
}
```

is actually a single statement. It clearly consists of two smaller statements, but from Java's point of view it is also a single compound statement. As a trivial case, we can also turn a single statement into a compound statement as in

```
{
    entry.setText( entry.getText() + clickedButton.getText() );
}
```

Thus, the `if` statement

```
if ( clickedButton == clearButton )
{
    total = 0;
    totalDisplay.setText( "0" );
}
else
{
    entry.setText( entry.getText() + clickedButton.getText() );
}
```

and all the other `if` statements we have shown in this chapter actually follow Java's rule that we should place a **single** statement before and after the word `else`. In these statements, however, the "single" statements used as the components of the `if` statements are all compound statements.

In some of the `if` statements we have shown, there have been examples where the curly braces included are not essential. The statement shown above, for example, could be rewritten as

```
if ( clickedButton == clearButton )
{
    total = 0;
    totalDisplay.setText( "0" );
}
else
    entry.setText( entry.getText() + clickedButton.getText() );
```

without changing its meaning in any way. Technically, wherever we have placed curly braces around a single statement, removing those braces will not change the behavior of the program. However, in all but one situation, we *strongly* encourage you to include curly braces around the statements you place in `if` statements. Doing so decreases the likelihood that you will accidentally make a fairly common programming mistake.

Often, in a context where you originally thought you only needed to put a single statement in the `else` part of an `if` statement, you later discover that you need to add more statements. If you are careful, you will always remember to add the needed curly braces while adding the extra statements. It is very easy, unfortunately, to forget to add the curly braces. Java typically can't identify this as an error. It simply fails to understand that you intended to include the additional statements as part of the `else`. Instead, it views them as independent statements that should always be executed after completing the execution of the `if` statement. Such errors can be hard to find, particularly if you use indentation appropriately to suggest your intended meaning while entering the additional statements. Java ignores the indentation, but you and anyone trying to help you understand why your program isn't working are likely be confused.

5.3.1 Multi-way Choices

There is one context where we do suggest that you leave out unnecessary curly braces. We suggest you do this when

- the curly braces surround an `if` statement that appears as the `else` branch of a larger `if` statement, and

```
if ( clickedButton == addButton ) {
    total = total + Integer.parseInt( entry.getText() );
    totalDisplay.setText( "" + total );
    entry.setText( "" );

} else if ( clickedButton == clearButton ) {
    total = 0;
    totalDisplay.setText( "0" );

} else {
    entry.setText( entry.getText() + clickedButton.getText() );

}
```

Figure 5.8: Formatting a multi-way choice

- the intent of the nested collection of `if` statements is to select one of three or more logically related alternatives.

In this case, eliminating the extra curly braces helps lead to a style of formatting that can make it easier for a reader to discern that the code is making a multi-way choice.

As an example, consider the 3-way choice made by the statement shown in Figure 5.7. If we remove the curly braces around the inner `if` statement, and then adjust the indentation so that the statements executed in all three cases are indented by the same amount, we obtain the code shown in Figure 5.8. Java considers these two versions of the code absolutely equivalent to one another. If you develop the habit of structuring multi-way choices as shown in Figure 5.8, however, you will find that it is easier to identify the choices being made and the code that goes with each choice than it is when the statements are formatted in a way that reflects the nesting of their grammatical structure.

Note that we have not removed all of the optional curly braces in Figure 5.8. We have only removed the curly braces surrounding nested `if` statements. The final curly braces could also be removed if our goal was to minimize the number of curly braces used, but all we want to do is remove enough braces to clarify the structure of the multi-way choice make by the nested `if` statements.

5.4 Something for Nothing

All of the `if` statements we have seen have been used to choose between several alternative sequences of statements. In certain programs, we use `if` statements to make a different kind of choice. In these programs we need to choose between doing something or doing nothing. For such programs, Java provides a second form of `if` statement in which the word `else` and the statement that follows is omitted. The grammatical rule for such `if` statements is

```
if ( condition )
    statement
```

Given our advice on the use of curly braces, however, we hope that when you write such `if` statements they will have the form

```
    if ( condition ) {
         sequence of statements
    }
```

As an example of the use of this form of `if` statement, let's think about a simple problem with the current version of our adding program. Suppose that as soon as we start running the program we decide to test every button on our program's keypad. With this in mind, we key in the number 9876543210 and press the "Add to total" button. We expect that this large number will show up in the "Total" field. In reality, however, what happens is that the development environment we are using pops up a new window displaying a rather mystifying collection of messages that starts with something that looks like the text below.

```
java.lang.NumberFormatException: For input string: "9876543210"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
        at java.lang.Integer.parseInt(Integer.java:480)
        at java.lang.Integer.parseInt(Integer.java:518)
        at AddingMachine.buttonClicked(AddingMachine.java:65)
                .
                .
                .
```

This ugly text is an example of a run-time error message. The computer is trying to tell us that something has gone wrong with our program. Even if we cannot understand every detail in this message, a few key words that appear in the message will give us some good clues.

The first line of the message describes the problem as a "NumberFormatException" (read as "number format exception") for the string 9876543210. The third and fourth lines mention the `Interger.parseInt` method we use in our program to convert `String`s into `int` values. Apparently, something went wrong when we tried to convert the string 9876543210 to an `int`.

The most common situation in which one encounters a number format exception is when your program tries to use `parseInt` to convert a string that isn't a number into one. For example, it is clear that the invocation

```
    Integer.parseInt( "think of a number" )
```

should be treated as an error. Unlike `"think of a number"`, however, the text `"9876543210"` certainly looks like a valid number. In fact, it looks like a very big number. That is the source of the problem. The number 9876543210 is actually too big for `parseInt`.

To allow us to manipulate numbers in a program, a computer's hardware must encode the numbers in some electronic device. For each digit of a number there must be a tiny memory device to hold it. Some fixed, finite number of these devices has to be set aside to represent each number. If a number is too big for its digits to fit in the memory devices set aside for it, the computer cannot handle the number. In Java, the number of devices set aside for an `int` is too small to handle the number 9876543210. Therefore, the computer has to treat the attempt to convert a string describing this large number as an error.

In a computer's memory, numbers are stored in binary, or base 2. Thirty-one binary digits are used to store the numeric value of each `int`. Just as the number of digits included in a car's odometer limits the range of mileage values that can be displayed before the odometer flips back to “000000”, the number of binary digits used to represent a number limit the range of numbers that can be stored. As a result, the values that can be processed by Java as `ints` range from -2,147,483,648 ($= -2^{31}$) to 2,147,483,647 ($= 2^{31} - 1$). If you try to use a number outside this range as an `int` value, Java is likely to complain that your program is in error or, worse yet, throw away some of the digits yielding an incorrect result.

Real calculators cannot display ugly messages like the one shown above if a user tries to enter too many digits. Instead, they typically just limit the size of the numbers that can be entered based on the number of digits included in the calculator’s display. If there are only 9 digits in the display, the typical calculator just ignores you if you try to enter more than 9 digits. We can solve our problem by making our adding machine behave in the same way.

In order to make the program ignore button clicks if the entry is already 9 digits long, we first need to change our program so that it keeps track of how many digits are currently in `entry`. In Figure 2.10 we showed a program that used an integer variable named `numberOfClicks` to keep track of how often a button had been clicked. We can keep track of how many digits are in the `entry` field by using a similar technique to count the clicks of the buttons of our adding machine program. For this purpose, we will introduce a variable named `digitsEntered`. We will add one to the value of this variable each time a digit key is pressed and assign the value 0 to `digitsEntered` when the entry field is cleared because the “Add to total” button was pressed.

To make it impossible for a person using our program to enter more than 9 digits, we can now simply place the code that adds digits to the `entry` field in an `if` statement that tests whether the value of `digitsEntered` is still less than 9. This `if` statement will choose between adding a digit to the entry field and doing nothing at all. Therefore, it will have no `else` part. The resulting statement will look like:

```
if ( digitsEntered < MAX_DIGITS ) {
    entry.setText( entry.getText() + clickedButton.getText() );
}
```

This code assumes that our program contains the instance variable declaration

```
private final int MAX_DIGITS = 9;
```

to associate the name `DISPLAY_SIZE` with the maximum number of digits allowed. It also uses a new form of condition. Java allows us to form conditions by comparing values. We have already seen that we can ask if two values are equal using the symbol `==`. When working with numbers, we can also use the symbols `<`, `>`, `<=`, and `>=` to test for “less than”, “greater than”, “less than or equal to”, and “greater than or equal to”, respectively. These symbols are called *relational operators.*

Java will also produce an error message if we apply `Integer.parseInt` to the empty string. This will happen if the user clicks “Add to total” twice in a row. To avoid this error, we should ignore any click on the “Add to total” button that occurs when the number of digits entered is zero.

The body of the complete `buttonClicked` method modified to make sure that the number of digits entered falls between 1 and 9 is shown in Figure 5.9. Notice that we assign zero to `digitsEntered` in the code that is executed when `addButton` is pressed and increase this variable’s value by one whenever a digit button is pressed.

```
public void buttonClicked( JButton clickedButton ) {
     if ( clickedButton == addButton ) {
         if ( digitsEntered > 0 ) {
             total = total + Integer.parseInt( entry.getText() );
             totalDisplay.setText( "" + total );
             entry.setText( "" );
             digitsEntered = 0;
         }

     } else if ( clickedButton == clearButton ) {
             total = 0;
             totalDisplay.setText( "0" );

     } else {
             digitsEntered = digitsEntered + 1;
             if ( digitsEntered < MAX_DIGITS ) {
                 entry.setText( entry.getText() + clickedButton.getText() );
             }
     }
 }
```

Figure 5.9: A `buttonClicked` method that ignores button clicks if `entry` is full or empty

5.5 Synonyms, Homographs, and Scopes

English contains many words that share common meanings. Groups of such words are known as synonyms. For example, while you might think of yourself as a student, others might call you a pupil, or a trainee.

In English, there are not only words that mean the same thing even though they are spelled differently, there are also words that mean different things even though they are spelled the same! Such words are called *homographs*. Consider the word "left". In certain contexts, this word refers to a side of the body as in "Take a left turn." In other context, it describes the action of departing. For example, you might say "Has everyone left?" Interestingly, in addition to describing the action of departing, "left" can be used to describe the action of remaining as in "Is there any food left?" Thus, in some sense, instead of talking about the single word "left" it is fair to talk about the three words "left", "left", and "left"!

It is also possible to have two names that are spelled exactly the same but have different meanings in a Java program. Determining the correct meaning to associate with a name that has multiple meanings in a Java program is a bit simpler than determining the correct meaning to associate with an English homograph like "left". It depends strictly on the context in which the use of the name occurs. A good analogy might be the use of a name like "the president" in English. If one of your classmates says "Now I am the president," she might have been chosen as president of the class or president of the chess club. On the other hand, if someone named Bush or Clinton said the same thing, the secret service would probably be close behind. The context required to correctly interpret the name "the president" is the social group involved (i.e., the chess club, the senior class, or the entire country).

The Java equivalent of the social groups that provide context in “the president” analogy are called *scopes*. The entire text of a Java class definition forms a scope. The constructor and each of the methods within a class are considered separate scopes. Note that the scopes corresponding to the constructor and the methods are also part of the larger scope corresponding to the entire class, just as a social group like the senior class may be part of another, larger group like all students on a campus or all residents of a certain country.

Within Java, curly braces indicate scope boundaries. The scopes we have already described, class and method bodies, are always enclosed in curly braces. In addition, within a method's body, each collection of statements and local variable declarations surrounded by their own pair of curly braces forms a smaller scope.

To clarify these ideas, Figure 5.10 shows the skeleton of our adding machine program with each of its distinct scopes highlighted by an enclosing rectangle. The largest rectangle corresponds to the scope associated with the entire class. The constructor and the `buttonClicked` methods are surrounded with rectangles to indicate that their bodies are both parts of the scope of the entire class and also represent smaller, more local scopes. Within the `buttonClicked` method the body of each of the `if` statements used is enclosed in curly braces. Therefore, we have also included rectangles to indicate that each of these collections of statements forms a scope. The only subtle point to notice about this figure is that the scopes corresponding to classes, methods and constructors include not just the text between the curly braces, but also the header lines that precede the opening curly braces.

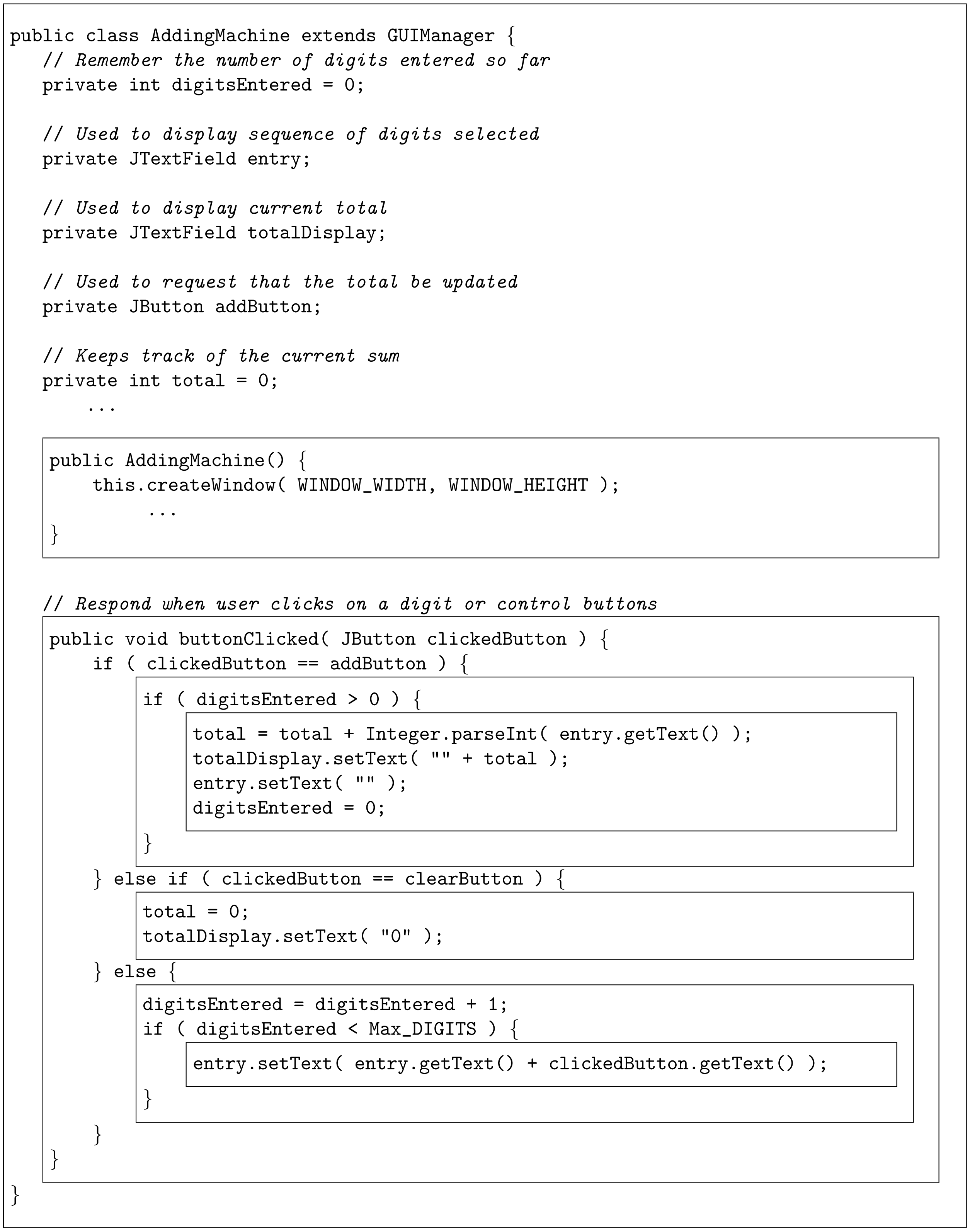
When a name is used within a Java program, the computer determines how to interpret the name by looking at the structure of the scopes within the program. We have already emphasized that every name that is to be used in a Java program must be declared somewhere in the program. In fact, a Java program must conform to a slightly stricter rule. Each use of a name within a Java program must be contained within the smallest scope that contains the declaration of the name.

Applying this rule to the program skeleton shown in Figure 5.10 is actually not very interesting because almost all of the names used in the program are declared as instance variables. The smallest scope that contains an instance variable's declaration is the scope of the entire class. Therefore, all references to instance variables within the program satisfy the “stricter” requirement trivially.

The one name in the program that is not declared in the scope of the full class is the formal parameter name `clickedButton`. The smallest scope containing the declaration of `clickedButton` is the scope of the `buttonClicked` method. All of the uses of this name fall within the same scope and are therefore valid. Suppose, however, that the programmer tried to include the statement

```
clickedButton.setEnabled( false );
```

within the body of the constructor for this class. This statement would violate Java's rule for the declaration of names because this use of the name would not be enclosed within the smallest scope that contains the declaration of the name `clickedButton` (i.e., the scope of the `buttonClicked` method). If you tried to compile such a program, you would receive an error message warning you that the reference to `clickedButton` within the constructor was invalid. This makes sense. Trying to use the value associated with the name `clickedButton` within the constructor would be silly since the computer cannot associate a meaning with this name until a button is actually clicked and the buttons will not even appear on the screen until the execution of the statements in the constructor are complete.



```
public class AddingMachine extends GUIManager {
   // Remember the number of digits entered so far
   private int digitsEntered = 0;

   // Used to display sequence of digits selected
   private JTextField entry;

   // Used to display current total
   private JTextField totalDisplay;

   // Used to request that the total be updated
   private JButton addButton;

   // Keeps track of the current sum
   private int total = 0;
       ...

   public AddingMachine() {
       this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );
            ...
   }

   // Respond when user clicks on a digit or control buttons
   public void buttonClicked( JButton clickedButton ) {
       if ( clickedButton == addButton ) {
            if ( digitsEntered > 0 ) {
                 total = total + Integer.parseInt( entry.getText() );
                 totalDisplay.setText( "" + total );
                 entry.setText( "" );
                 digitsEntered = 0;
            }
       } else if ( clickedButton == clearButton ) {
            total = 0;
            totalDisplay.setText( "0" );
       } else {
            digitsEntered = digitsEntered + 1;
            if ( digitsEntered < Max_DIGITS ) {
                 entry.setText( entry.getText() + clickedButton.getText() );
            }
       }
   }
}
```

Figure 5.10: Structure of declaration scopes within adding machine program

The existence of scopes becomes more interesting in programs that contain multiple declarations of a single name. In such programs, names can behave as homographs. That is, a single name can have distinct meanings in different scopes.

To illustrate this, suppose that we decided that we could make the code in the `buttonClicked` method a bit clearer by associating a local variable name with the contents of the `entry` text field. That is, suppose that we replaced the code in the first branch of the three-way `if` statement within `buttonClicked` with the following code:

```
if ( digitsEntered > 0 ) {
    String entryDigits;
    entryDigits = entry.getText();
    total = total + Integer.parseInt( entryDigits );
    totalDisplay.setText( "" + total );
    entry.setText( "" );
    digitsEntered = 0;
}
```

The first line in the body of this new `if` statement declares `entryDigits` as a local variable, the second line associates the contents of the `entry` text field with this name, and the third line references this new local variable. Admittedly, the introduction of the name `entryDigits` does not significantly improve the readability of this code. That is not the point. The goal of introducing this name is to present an example of reasonable code that leads to unreasonable problems if just a few more changes are made.

To see how problems can arise, suppose that while choosing a name for the local variable we didn't think about what other names were already being used in the program and decided to use the name `digitsEntered` instead of `entryDigits` to refer to the contents of the text field. A skeleton of the key parts of the program showing this change and the scopes involved can be found in Figure 5.11. Note that in this version of the program the name `digitsEntered` is declared twice — once as an instance variable and once as a local variable within the `if` statement we just modified.

In addition to containing the declaration

```
String digitsEntered;
```

the scope associated with the body of the `if` statement that handles the "Add to total" button now contains three references to the variable named `digitsEntered`. These occur in the two lines immediately following the declaration and in the last line of the body of the `if` statement:

```
digitsEntered = 0;
```

This last reference to `digitsEntered` was part of the original version of the program. It is supposed to set the value of the *instance variable* named `digitsEntered` back to 0 after an addition is performed. Unfortunately, in the revised program, Java will not interpret this line as a reference to the instance variable. This is because each reference to a name in a Java program is associated with the declaration of that name found in the smallest scope that contains the reference in question. The statement to set `digitsEntered` to 0 occurs within the same scope as the declaration of the local variable named `digitsEntered`. Therefore, this reference to `digitsEntered` will be associated with the local variable declaration. On the other hand, the references to `digitsEntered` that appear

```
public class AddingMachine extends GUIManager {
   // Remember the number of digits entered so far
   private int digitsEntered = 0;
       ...

   // Respond when user clicks on a digit or control buttons
   public void buttonClicked( JButton clickedButton ) {
       if ( clickedButton == addButton ) {
           if ( digitsEntered > 0 ) {
               String digitsEntered;
               digitsEntered = entry.getText();
               total = total + Integer.parseInt( digitsEntered );
               totalDisplay.setText( "" + total );
               entry.setText( "" );
               digitsEntered = 0;
           }
       } else if ( clickedButton == clearButton ) {
           total = 0;
           totalDisplay.setText( "0" );
       } else {
           digitsEntered = digitsEntered + 1;
           if ( digitsEntered < Max_DIGITS ) {
               entry.setText( entry.getText() + clickedButton.getText() );
           }
       }
   }
}
```

Figure 5.11: Incorrect code using a name as both a local and instance variable

elsewhere in the program are associated with the declaration of `digitsEntered` as an instance variable. For example, the statement

```
    digitsEntered = digitsEntered + 1;
```

that appears in the third branch of the main `if` statement of the `buttonClicked` method will increase the value associated with the instance variable.

In this program, therefore, the name `digitsEntered` behaves like an English homograph. It has different meanings in different contexts. In some sense, the local variable `digitsEntered` is a completely different name from the instance variable `digitsEntered`. Even though they are spelled the same, one name refers to a `String` while the other refers to an `int`.

As a consequence of the misinterpretation of the reference to `digitsEntered` within the statement

```
digitsEntered = 0;
```

the program whose skeleton is shown in Figure 5.11 will not compile correctly. Since the local variable `digitsEntered` is declared to refer to a `String` value, the Java compiler will be unhappy with this assignment that tries to associate it with the numeric value 0 instead of with a `String`. If you try to run this program, the Java compiler will report an error on this line and refuse to run the program.

Things could be worse.

Suppose that instead of using the local variable `digitsEntered` to refer to the `String` in the field named `entry`, we decided to make it refer to the value produced by applying `Integer.parseInt` to that `String`. In this case, we would use the following code for the `if` statement:

```
 if ( digitsEntered > 0 ) {
     int digitsEntered;
     digitsEntered =  Integer.parseInt( entry.getText() );
     total = total + digitsEntered;
     totalDisplay.setText( "" + total );
     entry.setText( "" );
     digitsEntered = 0;
 }
```

These changes won't have any impact on the way Java's scope rules associate uses of the name `digitsEntered` with the two declarations of the name. The reference to `digitsEntered` in the statement

```
digitsEntered = 0;
```

will still refer to the local variable rather than the instance variable. Now, however, since the local variable is declared as an `int`, the Java compiler will see nothing wrong with the code. The program will compile correctly. Unfortunately, it won't work as intended. Since the final assignment sets the local variable to 0 rather than setting the instance variable to 0, the value of the instance variable will never be reset to 0. Once more than 9 digits have been entered, the adding machine program will refuse to let its user enter any more digits. Such an error can be difficult to diagnose.

The best way to avoid this problem is to use different names for the local variable and the instance variable. To refine your understanding of scope rules, however, we want to point out that there is another way we could change the code so that it would work correctly even though both variables still had the same name. The key is to simply move the assignment that resets `digitsEntered` to 0 out of the body of the `if` statement in which the local variable is declared. A skeleton of the code for the program that would result is shown in Figure 5.12. Now, the smallest scope that contains both the assignment and a declaration of the name `digitsEntered` is the scope of the entire class. Therefore, Java will associate this reference to `digitsEntered` with the instance variable declaration of the name as desired.

5.6 Summary

When we introduced Java in Chapter 1, we suggested that the process of learning Java or any new language typically involved learning bits of grammar and bits of vocabulary concurrently. This chapter has been something of an exception to that rule. In this chapter, we focused primarily on grammatical forms.

We learned two ways in which Java statements can be combined to form larger, compound statements. Our primary focus was on the `if` statement, a grammatical construct used to express commands that require a program to choose between two alternative execution paths. We learned that there are two forms of `if` statements. One form is used to describe situations where the program needs to choose whether or not to execute a sequence of one or more statements. The other form is used when a program needs to choose between two distinct sets of statements.

Although Java's grammatical rules only explicitly provide forms of the `if` statement targeted at simple 2-way choices, we learned that these forms were flexible enough to let us express more complex, multi-way choices. The key to this is the fact that we can use one `if` statement as a part of a larger `if` statement forming what is called a nested `if` statement.

With the introduction of `if` statements, the number of curly braces in our programs increased dramatically. In our examples, curly braces were placed around the sequences of statements whose execution was controlled by an `if` statement. We learned that under Java's grammatical rules, such curly braces were actually part of a very simple form of compound statement. By simply surrounding a sequence of statements with curly braces we tell Java that we want those statements to be treated as a single, compound statement.

Finally, the introduction of `if` statements involved the introduction of conditions used to tell Java how to make choices when executing an `if` statement. The conditions we used in this chapter all involved telling Java to compare values either to see if two names referred to the same object or to see if one number was larger than another. We will see in a few chapters that many other forms of conditions can be used in `if` statements and in other contexts.

```
public class AddingMachine extends GUIManager {
   // Remember the number of digits entered so far
   private int digitsEntered = 0;
       ...

   // Respond when user clicks on a digit or control buttons
   public void buttonClicked( JButton clickedButton ) {
       if ( clickedButton == addButton ) {
           if ( digitsEntered > 0 ) {
               String digitsEntered;
               digitsEntered = entry.getText();
               total = total + Integer.parseInt( digitsEntered );
               totalDisplay.setText( "" + total );
               entry.setText( "" );
           }
           digitsEntered = 0;
       } else if ( clickedButton == clearButton ) {
           total = 0;
           totalDisplay.setText( "0" );
       } else {
           digitsEntered = digitsEntered + 1;
           if ( digitsEntered < Max_DIGITS ) {
               entry.setText( entry.getText() + clickedButton.getText() );
           }
       }
   }
}
```

Figure 5.12: Correct (but unpleasant) code using a name as both a local and instance variable