# Chapter 7

# Primitive Technology

Early applications of computers, from tabulating census data to calculating projectile trajectories, focused primarily on numeric calculations. By contrast, the most complicated calculation we have considered thus far was to count how often a button had been clicked. This reflects the fact that current applications of computers are much less numerically oriented than their predecessors. Nevertheless, some basic familiarity with the techniques used to work with numbers in a program is still essential.

In this chapter, we will explore Java's mechanisms for numeric computation. We will learn more about the type `int` and that there are several types other than `int` that can be used to represent numeric values in Java. We will see that all of these types, which Java classifies as *primitive* types, are different in certain ways from types defined as classes.

Finally, we will explore Java's only non-numeric primitive type, `boolean`. This type contains the values used to represent the results of evaluating conditions found in `if` statements.

## 7.1 Planning a Calculator

In Chapter 5, we showed how to write a simple adding machine program. The adding machine we developed in that chapter isn't very impressive. In fact, it is a bit embarrassing. Any calculator you can buy will be able to at least perform addition, multiplication, subtraction, and division. If you pay more than $10 for a calculator, it will probably include trigonometric functions. Meanwhile, all our efforts have produced is a program that will enable your computer, which probably costs at least 100 times as much as a basic calculator, to do addition.

In this chapter, we will extend our adding machine program to provide the functionality of a simple, inexpensive calculator. Our goal will be a program that can add, multiply, subtract, and divide.

Constructing a program that simulates a calculator requires a bit of thought because most calculators are a bit more subtle than our adding machine. As we did for our adding machine program, we provide a series of screen images showing how we would expect our calculator to perform in Figure 7.1. The behavior illustrated in this figure is identical to what one would expect from any basic calculator.

The image in the upper left corner of the figure shows how the calculator's interface would appear just after the program begins to run. The constructor definition that creates this interface is shown in Figure 7.2 and the beginning of the class definition for such a program including all

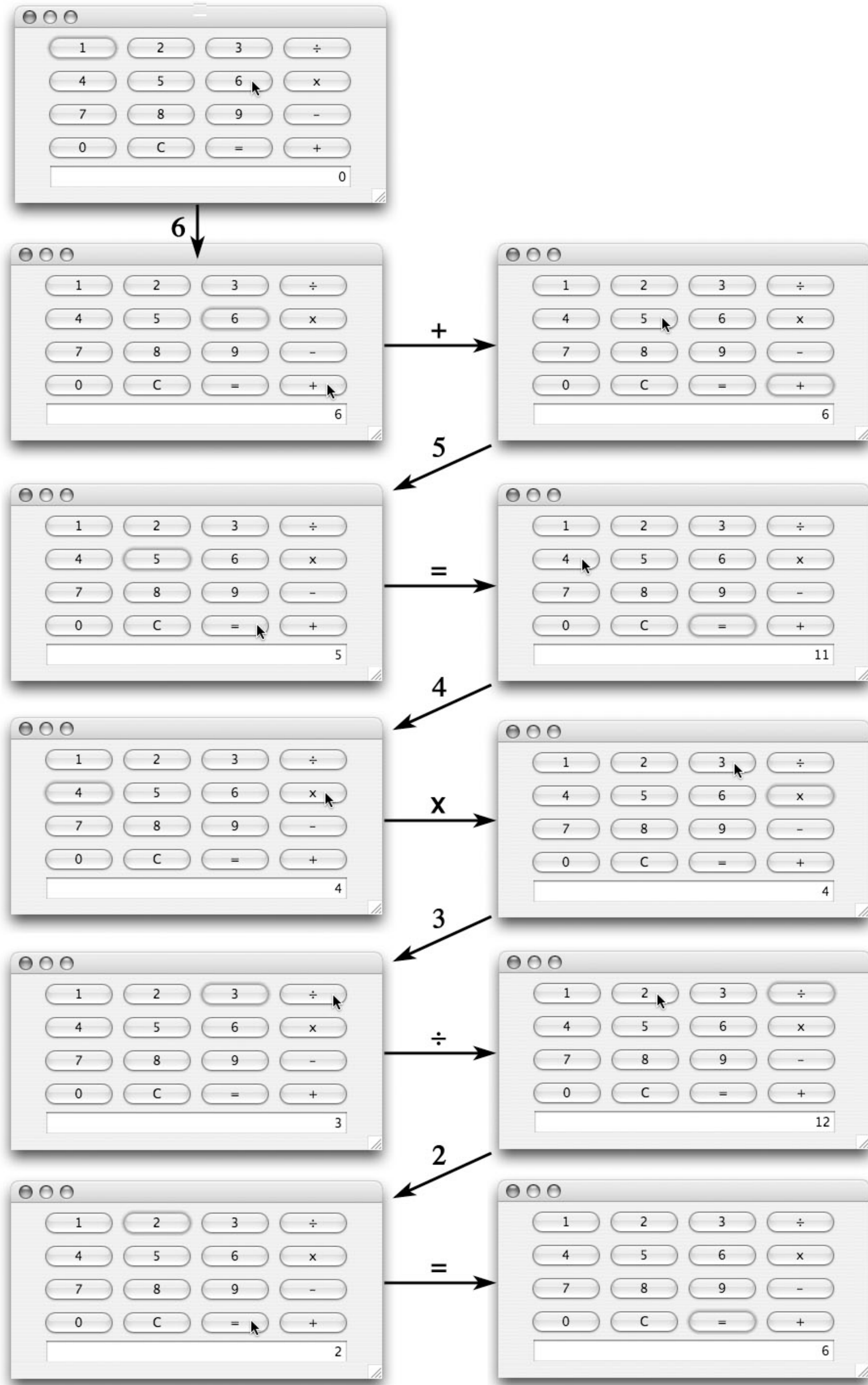


Figure 7.1: Interface for a 4-function calculator program

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

    contentPane.add( new JButton( "1" ) );
    contentPane.add( new JButton( "2" ) );
    contentPane.add( new JButton( "3" ) );
    contentPane.add( divide );

    contentPane.add( new JButton( "4" ) );
    contentPane.add( new JButton( "5" ) );
    contentPane.add( new JButton( "6" ) );
    contentPane.add( times );

    contentPane.add( new JButton( "7" ) );
    contentPane.add( new JButton( "8" ) );
    contentPane.add( new JButton( "9" ) );
    contentPane.add( minus );

    contentPane.add( new JButton( "0" ) );
    contentPane.add( clear );
    contentPane.add( equals );
    contentPane.add( plus );

    entry.setHorizontalAlignment( JTextField.RIGHT );
    contentPane.add( entry );
}
```

Figure 7.2: The constructor for a simple calculator program

of the instance variables used in the constructor is shown in Figure 7.3. The only new feature exhibited in this code is the invocation of a method named `setHorizontalAlignment` in the next to last line of the constructor. This invocation causes the value displayed in the calculator's text field to be right justified within the field.

The first five images in in Figure 7.1 show the program being used to perform the calculation "6 + 5 = 11". This is accomplished by entering "6 + 5 =" on the calculator's keypad with the mouse. The important thing to notice is that the "+" key on this calculator is not simply a replacement for the "Add to total" button in our adding machine program. When the "+" key is pressed, nothing actually changes in the program's window The calculator cannot perform the addition because the second operand, 5 in this example, has not even been entered. Instead, the program waits until the second operand has been entered and the "=" key is pressed. At that point, it adds the two operands together. The "=" key, however, is also more than a replacement for the "Add to total" button. Pressing it does not always cause the program to add its operands. The operation performed depends on which keys had been pressed earlier. If the user had entered "6 - 5 =", then the program should perform a subtraction when the "=" key is pressed. The program must

```
// A $5 Calculator
public class IntCalculator extends GUIManager {
   // Change these values to adjust the size of the program's window
   private final int WINDOW_WIDTH = 380, WINDOW_HEIGHT = 220;

   // Maximum number of digits allowed in entry
   private final int MAX_DIGITS = 9;

   // Used to display sequence of digits selected and totals
   private JTextField entry = new JTextField( "0", 25 );

   // Operator buttons
   private JButton plus = new JButton( "+" );
   private JButton minus = new JButton( "-" );
   private JButton times = new JButton( "x" );
   private JButton divide = new JButton("÷" );
   private JButton equals = new JButton( "=" );

   // The clear button
   private JButton clear = new JButton( "C" );

   // Remember the number of digits entered so far
   private int digitsEntered = 0;

   // Value of current computation
   private int total = 0;
```

Figure 7.3: Instance variable declarations for a simple calculator program

somehow remember the last operator key pressed until the "`=`" key is pressed.

The remaining screen images in Figure 7.1 show how the program should behave if after computing "6 + 5 = 11", the person using the calculator immediate enters "4 x 3 ÷ 2 =". The interesting thing to notice is that when the user presses the ÷ key, the calculator displays the result of multiplying 4 and 3. This is exactly what the calculator would have done if the user had pressed the "`=`" key instead of the ÷ key at this point. From this we can see that the "`=`" key and all of the arithmetic operator keys behave in surprisingly similar ways. When you press any of these keys, the calculator performs the operation associated with the last operator key pressed to combine the current total with the number just entered. The only tricky cases are what to do if there is no "last operator key pressed" or no "number just entered".

The no "last operator key pressed" situation can occur either because the program just started running or because the last non-numeric key pressed was the "`=`" key. In these situations, the calculator should simply make the current total equal to the number just entered.

The no "number just entered" situation occurs when the user presses two non-numeric keys in a row. Real calculators handle these situations in many distinct ways. For example, on many calculators, pressing the "`=`" key several times causes the calculator to perform the last operation entered repeatedly. We will take a very simple approach to this situation. If the user presses several non-numeric keys in a row, our calculator will ignore all but the first non-numeric key pressed.

From this description of the behavior our calculator should exhibit, we can begin to outline the structure of the `if` statements that will be used in the `buttonClicked` method. Looking back at the code in the `buttonClicked` method of the last version of our adding machine program provides a good starting point. That code is shown in Figure 5.9. The `if` statement used in that method implemented a three-way choice so that the program could distinguish situations in which the user had pressed the "Clear Total" button, the "Add to Total" button, or a numeric key. From the discussion above, we can see that our calculator program must make a similar three-way choice. The cases it must distinguish are whether the user has pressed the clear button, a numeric key, or any of the other non-numeric keys (i.e., an operator key or the "`=`" key). In the case that the user has pressed an operator key, it needs to make a second decision. If no digits have been entered since the last operator key was pressed, it should ignore the operator. Based on these observations, we can sketch an outline for this program's `buttonClicked` method as shown in Figure 7.4.

Obviously, we cannot type a program containing code like the outline shown in this figure into our IDE and expect to run it. We can, however, type such an outline in and then use it as a template in which we "fill in the blanks" as we complete the program. With this in mind, we have indicated the places in our outline where detailed code is missing using ellipses preceded by comments explaining what the missing code should do.

Writing such an outline can be a very useful step in the process of completing a program. Much as outlining a paper allows you to work out the overall organization of your thoughts without worrying about the precise wording you will use for every sentence, outlining a program enables you to focus on the high-level structure of your program. With a good understanding of this structure, it is generally much easier to fill in the many details needed to complete a program.

## 7.2 Smooth Operators

One detail we obviously need to explore to complete our calculator is how to perform arithmetic operations other than addition. Performing arithmetic operators in Java is actually quite simple.

```
// Accept digits entered and perform arithmetic operations requested
public void buttonClicked( JButton clickedButton ) {
     if ( /* clickedButton is an operator or the = key */ ...  ) {
         if ( /* the last key pressed was not an operator or = key */ ... ) {
            // Apply last operator to the number entered and current total
                 . . .
         }

     } else if ( clickedButton == clearButton ) {
        //  Zero the total and clear the entry field
                . . .

     } else { // If clickedButton is a numeric key, add  digit to entry
                . . .
     }
 }
```

Figure 7.4: Outline for the `buttonClicked` method of a calculator program

In addition to the plus sign, + Java recognizes operators for the other three standard arithmetic operations. The minus sign, -, is used for subtraction, - the slash, /, is used for division, and the asterisk, *, is used for multiplication. Thus, just as we used the statement

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

in our adding machine program, we can use a statement like

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

to perform subtraction[1], or statements like

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

and

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

to perform multiplication and division. * /

As we noted in the previous section, the operation performed by our calculator when an operator key or the "=" key is pressed is actually determined by which of these keys was the last such key pressed. Therefore, we will have to include an instance variable in our class that will be used to keep track of the previous operator key that was depressed. We can do this by adding a declaration of the form

```
// Remember the last operator button pressed
private JButton lastOperator = equals;
```

to those already shown in Figure 7.3. We will use this variable in the `buttonClicked` method to decide what operation to perform. We will have to include code in the `buttonClicked` method to change the value of `lastOperator` each time an operator key is pressed. We have initialized `lastOperator` to refer to the `equals` key because when it first starts to run we want our calculator to behave as it does right after the "=" key is pressed.

The code required to use `lastOperator` in this way is included in Figure 7.5. The code we have added in this figure begins by using `Integer.parseInt` to convert the digits entered by the user into a number. Then, we use a series of `if` statements to make a 5-way choice based on the previously pressed operator key by comparing the value of `lastOperator` to the names that refer to each of the operator keys. Each of the five lines that may be selected for execution by these `if` statements updates the value of the `total` variable in the appropriate way. Next, we display the updated value of `total` in the `entry` text field. Finally, but very importantly, we use an assignment statement to tell the computer to associate the name `lastOperator` with the button that was just pressed. This ensures that the next time an operator key is pressed, the program will be able to perform the correct operation.

---

[1] In Java, the minus sign can also be used with a single operand. That is, if the value of `total` is `10`, then executing the statement

```
total = - total;
```

will change the value to `-10`.

```
// Accept digits entered and perform arithmetic operations requested
public void buttonClicked( JButton clickedButton ) {
     if ( /* clickedButton is an operator or the = key */ ...  ) {
         if ( /* the last key pressed was not an operator or = key */ ... ) {
            // Apply last operator to the number entered and current total

            int numberEntered = Integer.parseInt( entry.getText() );

            if ( lastOperator == plus ) {
                total = total + numberEntered;
            } else if ( lastOperator == minus ) {
                total = total - numberEntered;
            } else if ( lastOperator == times ) {
                total = total * numberEntered;
            } else if ( lastOperator == divide ) {
                total = total / numberEntered;
            } else {
                total = numberEntered;
            }

            entry.setText( "" + total );

            lastOperator = clickedButton;
                 . . .
         }

     } else if ( clickedButton == clearButton ) {
        //  Zero the total and clear the entry field
                . . .

     } else { // If clickedButton is a numeric key, add  digit to entry
                . . .
     }
 }
```

Figure 7.5: Refined outline for calculator `buttonClicked` method

### 7.2.1 Relational Operators and `boolean` Values

In the instance variable declarations for our calculator program (shown in Figure 7.3), we included a variable named `digitsEntered`. In the final version of our adding machine program we used a similar variable to keep track of the number of digits in the value the user was currently entering so that a user could not enter a value too big to be represented as an `int`. In our calculator program, this variable will serve two additional roles.

First, we indicated earlier that our program should ignore all but the first operator key pressed when several operator keys are pressed consecutively. We will use `digitsEntered` to detect such situations. When an operator key is pressed, our program "consumes" the most recently entered number. We can therefore determine whether two operator keys have been pressed in a row by checking whether the value of `digitsEntered` is 0 when an operator key is pressed.

Second, when the user presses an operator key, our program will display the current value of `total` in the `entry` field. This value should remain visible until the user presses a numeric key (or the clear button). As a result, our program must perform a special step when handling the first numeric key pressed after an operator key. It must clear the `entry` field. Again, we can use the value of `digitsEntered` to identify such situations.

A revised version of our outline of the `buttonClicked` method, extended to include the code that updates and uses `digitsEntered`, is shown in Figure 7.6. Actually, it is a bit of a stretch to still call this an "outline". The only detail left to be resolved is the condition to place in the `if` statement at the very beginning of the method.

In the branch of the method's main `if` statement that handles operator keys, we have added an assignment to associate 0 with the name `digitsEntered`. We have also filled in the branches of this `if` statement that handle the clear button and the numeric keys.

The code for the clear button sets the total and the number of digits entered to zero. It also clears the entry field. Finally, it lies just a little bit by associating the "`=`" key with `lastOperator`. After the clear key is pressed we want the program to behave as if the last operator key pressed was "`=`".

The code for numeric keys begins by clearing the `entry` text field if the value of `digitsEntered` is 0. That is, it clears the text field when a numeric key is pressed immediately after one of the non-numeric keys has been pressed. The rest of the code for handling numeric keys is identical to the code we used in our adding machine program.

Finally, we have replaced the comment in the `if` statement:

```
if ( /* the last key pressed was not an operator or = key */ ... ) {
```

with the condition

```
digitsEntered > 0
```

The symbols `==`, `!=`, `<`, `<=`, `>=`, and `>` are examples of what we call *relational operators*. The operator `>` can be used to make Java check whether one value is greater than another. The operator `>=` tells Java to check whether one value is greater than or equal to another. That is, `>=` is the equivalent of the mathematical symbol $\geq$. For example, as an alternative to the `if` statement

```
if ( digitsEntered > 0 ) {
```

we could have written

```
// Accept digits entered and perform arithmetic operations requested
public void buttonClicked( JButton clickedButton ) {
    if ( /* clickedButton is an operator or the = key */ ...  ) {
        if (  digitsEntered > 0  ) {
           // Apply last operator to the number entered and current total

           int numberEntered = Integer.parseInt( entry.getText() );

           if ( lastOperator == plus ) {
               total = total + numberEntered;
           } else if ( lastOperator == minus ) {
               total = total - numberEntered;
           } else if ( lastOperator == times ) {
               total = total * numberEntered;
           } else if ( lastOperator == divide ) {
               total = total / numberEntered;
           } else {
               total = numberEntered;
           }

           entry.setText( "" + total );
           digitsEntered = 0;

           lastOperator = clickedButton;
        }

    } else if ( clickedButton == clearButton ) {
       //  Zero the total and clear the entry field
       total = 0;
       digitsEntered = 0;
       entry.setText( "0" );
       lastOperator = equals;

    } else { // If clickedButton is a numeric key, add  digit to entry
       if ( digitsEntered == 0 ) {
           entry.setText( "" );
       }

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

Figure 7.6: Calculator outline extended with details of using `digitsEntered`

```
if ( digitsEntered >= 1 ) {
```

Similarly, the operator `<=` is used to check whether one value is less than or equal to another. Finally, the operator `!=` is used to ask if two values are different. That is, in Java, `!=` is equivalent to the mathematical symbol $\neq$.

The relational operators `==` and `!=` can be used to compare values described by any two expressions in a program.[2] The remaining relationals, `<`, `>`, `<=`, and `>=` can only be used to compare numeric values.

The fact that `<`, `>`, `==`, `<=`, `>=`, and `!=` are referred to as relational **operators** reflects an important fact about the way these symbols are understood within the Java language. The other operators we have discussed are the arithmetic operators `+`, `-`, `*`, and `/`. Within a Java program, phrases that involve arithmetic operators are identified as expressions. That is, they are used in contexts where the programmer needs to describe values. For example, `3 * 4` is an expression that describes the value `12`. We have also seen that even without knowing the exact value that an expression will produce in a Java program, we can always identify the type from which the value will come. For example, if `x` is an `int` variable, we know that the expression `x + 1` will describe an `int` value rather than a `String` or `JTextField`, even though we cannot say which `int` value the expression describes until we determine the current value of `x`. If the relational operators are to be interpreted in a similar way, then we have to consider phrases like `3 != 4` and `x > 0` as expressions and we have to be able to identify the type of value each such expression describes.

The value described by an expression like `x > 0` will not be another `int` or any of the other types we have encountered thus far. Instead, Java includes an additional type called `boolean` that consists of the values described by expressions involving relational operators. This type contains just two values described by the literals `true` and `false`. If the value associated with the variable `x` is 14, then we would informally say that it is true that `x` is greater than 0. In Java, therefore, if the value of `x` is 14, then the value described by the expression `x > 0` is `true`. On the other hand, if `x` is 0 or any negative number then the value described by `x > 0` is `false`.

We will see in the next few sections, that the values of the type `boolean` can be manipulated in many of the same ways we can manipulate values like `int`s. We can declare variables that will refer to `boolean` values. There are operators that take `boolean` values as operands. We can describe `boolean` values in our program using the literals `true` and `false` just as we describe `int` values by including literals like `1` and `350` in our code. In addition, the type `boolean` plays a special role in Java. Expressions that produce `boolean` values are used as conditions in `if` statements and other structures used to control the sequence in which a program's instructions are executed.

### 7.2.2 Primitive Ways

It is important to note that there are several ways in which Java treats `int`s differently from many other types of information we manipulate in our programs. When we want to perform an operation on a `JTextField` or a `JButton`, we use method invocations rather than operators. The designers of the Java language could have also provided methods for the operations associated with arithmetic and relational operators. That is, we might have been expected to type things like

```
x.plus(y)
```

[2]Note: Although `==` can be used to compare any two values, we will soon see that it is often more appropriate to use a method named `equals` to compare certain types of values.

rather than

```
x + y
```

or

```
x.greaterthan(0)
```

rather than

```
x > 0
```

Obviously, Java's support for operator symbols is a great convenience to the programmer. Operator notation is both more familiar and more compact than the alternative of using method invocations to perform simple operations on numbers. At the same time, the decision to provide operator symbols within Java represents a more fundamental difference between numbers and objects like `JTextField`s.

First, it is worth noting that within Java we can manipulate numbers using many of the same mechanisms that are used with GUI components and other objects. We can declare variables of type `int` just as we can declare variables to refer to `JButton`s and `JTextField`s. We have seen some examples where GUI components are passed as parameters (e.g., to `contentPane.add`) and others where numbers are passed (e.g., to `this.createWindow`). At the same time, numbers and objects like `JPanel`s and `JTextArea`s are treated differently in several important ways. As we just observed, operators are used to manipulate numbers while methods are used to manipulate GUI components and other objects. In addition, while we use constructions when we want to describe a new GUI component, Java doesn't make (or even let) us say

```
int x = new int( 3 );
```

These differences reflect that fact that Java views numbers as permanent and unchanging while GUI components and many other types of objects can be changed in various ways. In some oddly philosophical sense, Java doesn't let us say

```
int x = new int( 3 );
```

because it makes no sense to talk about making a new 3. The number 3 already exists before you use it. How would the new 3 differ from the old 3? On the other hand, Java will let you say

```
JLabel homeScore = new JLabel( "0" );
JLabel visitorsScore = new JLabel( "0" );
```

because each of the constructions of the form `new JLabel( "0" )` creates a distinct `JLabel`. They might look identical initially, but if we later execute an invocation such a

```
homeScore.setText( "2" );
```

it will become very clear that they are different.

This notion also underlies the distinction between operators and methods. Operations performed using method invocations often change an existing object. As we just showed, the use of `setText` in the invocation

```
homeScore.setText( "2" );
```

changes the state of the `JLabel` to which it is applied. On the other hand, if we declare

```
int x = 3;
```

and then evaluate the expression

```
x * 2
```

neither `x` nor `2` turn into 6. The expression tells the computer to perform an operation that will yield a new value, but it does not change the values used as operands in any way. This remains true even if we include the expression in an assignment of the form

```
x = x * 2;
```

The assignment statement now tells Java to change the meaning associated with the name `x` so that `x` refers to the value produced by the expression, but evaluating the expression itself does not change either of the two numbers used as operands to the multiplication operator.

This distinction becomes most obvious when two different names are associated with a single value. For example, suppose that instead of creating two distinct `JLabel`s for the scores of the home team and the visitors as shown above, we write the declarations

```
JLabel homeScore = new JLabel( "0" );
JLabel visitorsScore = homeScore;
```

This code creates a single `JLabel` and associates it with two names. If we then execute the invocation

```
homeScore.setText( "2" );
```

the computer will change this `JLabel` rather than creating a new `JLabel` with different contents. Since both names refer to this modified `JLabel` the invocations

```
homeScore.getText( );
```

and

```
visitorsScore.getText( );
```

will both produce the same result, `"2"`.

On the other hand, if we declare

```
int x = 3;
int y = x;
```

and then execute the assignment

```
x = x * 2;
```

the value of `x` will be changed to 6, but the value `y` refers to will still be 3. There are two reasons for this. First, the multiplication `x * 2` identifies a new value, 6 in this case, rather than somehow modifying either of its operand values to become 6. Second, you must realize that variable's refer to values, not to other variables. Therefore, when we say

```
int x = 3;
int y = x;
```

we are telling Java that we want `y` to refer to the same value that `x` refers to at the point that the computer evaluates the declaration of `y`. This will be the value 3. Later, executing the assignment

```
x = x * 2;
```

makes the name `x` refer to a new value, 6, but does not change the value to which `y` refers.

The integers are not the only Java type that has these special properties. The `boolean` values introduced in the preceding section have similar properties. You cannot create a "new" `true` or `false`. There are no methods associated with the type `boolean`, but, as we will see in the next section, there are several operators that can be used to manipulate `boolean` values. In addition, there are several other arithmetic types that behave in similar ways.

Java refers to pieces of information that we construct using `new` and can manipulate with methods as *objects* and the types composed of such items are called *object types* or *classes*. On the other hand, pieces of information that are described using literals and manipulated with operators rather than methods are called *primitive values* and types composed of primitive values are called *primitive types*. To make it easy for programmers to remember which types are primitive types and which are classes, the designers of Java followed the helpful convention of using names that start with lower-case letters for primitive types and names that start with upper-case letters for the names of classes.

### 7.2.3 Logical Arguments

There is a common mistake made by novice programmers when using relational operators. This mistake reveals an interesting difference between Java's relational operators and mathematical symbols like $\leq$ and $\geq$. In mathematics, it is common to describe the fact that a variable falls in a specific range by writing something like

$$1 \leq x \leq 10$$

As a result, new programmers sometimes write `if` statements that look like

```
if ( 1 <= x <= 10 ) {
```

Unfortunately, the Java compiler will reject the condition in such an `if` statement and display an error message that says:

```
operator <= cannot be applied to boolean,int
```

To understand why Java rejects conditions of this form, you have to remember that Java interprets symbols like `<=` as operators and applies them exactly as it applies `+`, `*`, and other arithmetic operators. If you knew that the name `x` was associated with the number 6 and were asked to describe the process you use to determine the value of the expression

```
1 + x + 10
```

you would probably say something like "1 plus 6 is 7 and 7 + 10 is 17 so the total is 17." When evaluating this expression, you first determine the result of applying `+` to the first two operands and then you use the result of that addition as the first operand of the second addition.

Java tries to take the same approach when evaluating

```
1 <= x <= 10
```

It first wants to determine the value described by

```
1 <= x
```

Assuming as we did above that `x` is associated with 6, this subexpressions describes the value `true`. Then, Java would try to use the result of this evaluation as the first operand to the second operator. It would therefore end up trying to determine the value of the expression

```
true <= 10
```

Unfortunately, you can't compare `true` to 10. It just doesn't make any sense. That is what Java is trying to tell you with its error message.

Although Java will reject an `if` statement with the condition in

```
if ( 1 <= x <= 10 ) {
```

it does provide a way to express the intent of such a statement. The mathematical notation

$$1 \leq x \leq 10$$

really combines two relationships that involve the value of $x$. It states that:

1 must be less than or equal to $x$
***and***
$x$ must be less than or equal to 10.

Java provides an operator that can be used to express the conjunction "and" in such a statement. The symbol used for this operator is a pair of consecutive ampersands, `&&`. Thus, a correct way to express this condition in an `if` statement is

```
if ( ( 1 <= x ) && ( x <= 10) ) {
```

The `&&` operator takes `boolean` values as operands and produces a `boolean` value as its result. Such operators are called *logical operators*. In addition to the operator `&&`, Java provides an operator that can be used to express conditions involving the conjunction "or". This operator is typed as two consecutive vertical bars, `||`.

The or operator provides the tool we need to complete the `buttonClicked` method for our calculator program. In our latest version of the outline for the calculator's `buttonClicked` method, the only aspect that still needs to be replaced with real code is the condition in the first `if` statement:

```
if ( /* clickedButton is an operator or the = key */ ...  ) {
```

The button clicked is an operator key only if it is either the "`+`" key, ***or*** the "`-`" key, ***or*** the "`*`" key, ***or*** the "" key, ***or*** the "`=`" key. We can use the `||` operator to express this as

```
if ( clickedButton == minus  || clickedButton == times ||
     clickedButton == divide || clickedButton == plus  ||
     clickedButton == equals ) {
```

```
// Accept digits entered and perform arithmetic operations requested
public void buttonClicked( JButton clickedButton ) {
     if ( clickedButton == minus  || clickedButton == times ||
          clickedButton == divide || clickedButton == plus  ||
          clickedButton == equals ) {
         if (  digitsEntered > 0  ) {
            // Apply last operator to the number entered and current total

            int numberEntered = Integer.parseInt( entry.getText() );

            if ( lastOperator == plus ) {
                total = total + numberEntered;
            } else if ( lastOperator == minus ) {
                total = total - numberEntered;
            } else if ( lastOperator == times ) {
                total = total * numberEntered;
            } else if ( lastOperator == divide ) {
                total = total / numberEntered;
            } else {
                total = numberEntered;
            }
            entry.setText( "" + total );
            digitsEntered = 0;
            lastOperator = clickedButton;
         }

     } else if ( clickedButton == clearButton ) {
        //  Zero the total and clear the entry field
        total = 0;
        digitsEntered = 0;
        entry.setText( "0" );
        lastOperator = equals;

     } else { // If clickedButton is a numeric key, add  digit to entry
        if ( digitsEntered == 0 ) {
            entry.setText( "" );
        }

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

Figure 7.7: The complete `buttonClicked` method for a calculator program

Java will consider this condition to be `true` if the name `clickedButton` refers to any of the five operator keys. This leads to a complete specification of the code for `buttonClicked` as shown in Figure 7.7.

In addition to `&&` and `||`, Java provides a third logical operator that expects just a single operand. This operator is represented using the exclamation point, `!`. This operator is read as the word "not". It produces the `boolean` value that is the opposite of its operand. That is, `!true` is `false` and `!false` is `true`. For example, the condition

```
! ( clickedButton != plus )
```

always produces the same result as

```
( clickedButton == plus )
```

We should be clear, however, that there is no good reason to ever write an expression like

```
! ( clickedButton != plus )
```

Simply using the `==` operator is simpler and clearer. Later, we will see that more useful applications of the `!` operator occur when we are using `boolean` variables or accessor methods that produce `boolean` values.

## 7.3 `double` Time

Our calculator is still lacking at least one feature found on even the cheapest calculators you can buy. There is no decimal point button. That means there is no direct way to enter a number like .5 or any other number with digits after the decimal point.

If you were trying to use our calculator to evaluate a formula that involved .5, you might imagine that you could still complete the calculation without a decimal point button by simply entering 1 divided by 2 where you wanted to use the value .5. If you tried this, however, you would be disappointed. If a user presses the keys `1`, $\div$, `2`, and `=` on our calculator, the program displays 0 as the answer, rather than .5.

The reason for this behavior is simple. Within our program, we have declared the variables `total` and `numberEntered` to refer to values of type `int`. The name `int` is short for *integer*, the set of numbers that have no fractional parts. When we tell Java that we want to work with integers, it ignores the fractional parts of any computations it performs for us. `1/2` becomes 0, `14/5` becomes 2, and so on. Our calculator doesn't just have no decimal point button in its keypad, it doesn't even know about decimal points.

Fortunately, it is quite easy to teach our calculator about decimal points. In addition to the type `int`, Java has another primitive type for numbers with fractional parts (what mathematicians call rational numbers). The odd thing is that this type is not named `rational` (or even `rat`). Instead, it is called `double`. We will try to explain the logic behind this odd name later. For now, we will ask you to just accept it and we will use it to quickly convert our program into a calculator that supports numbers other than integers. We can do this by simply replacing the name `int` with `double` in the declaration of the instance variable `total`. That is, the declaration for `total` would now look like:

```
// Value of current calculation
private double total = 0;
```

We have seen that Java interprets the + operator as concatenation if either operand is a String to convert int values into String representations. Fortunately, this property of the + operator extends to double values as well. As a result, after changing the declaration of total to make it a double variable, we can still use the instruction

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

to display its value. Therefore, once total's declaration is changed, our calculator will perform calculations that produce fractional values correctly. If a user enters "1 ÷ 2", the modified program will now display 0.5 as the result. We still won't be able to enter decimal points (we will save the task of supporting a decimal point key for the next section), but the results our calculator produces will at least be accurate.

### 7.3.1 Seeing double

While we can use the concatenation operator to convert double values into String form for display, the results of such conversions are sometimes a bit surprising.

First, whenever a double is converted to a String. Java includes a decimal point. If we declare

```
private int anInt;
private double aDouble;
```

and then associate these names with equivalent values by executing the assignments

```
anInt = 1000;
aDouble = anInt;
```

executing the command

```
entry.setText( anInt + " = " + aDouble );
```

will produce the display

```
1000 = 1000.0
```

Things get a bit more interesting when large values of type double are converted to String form. Suppose that rather than initializing the two variables above to 1000, we instead execute the assignments

```
anInt = 1000000000;
aDouble = anInt;
```

so that both variables have a value of one billion. In this case, the display produced will be

```
1000000000 = 1.0E9
```

The strange item "1.0E9" is an example of Java's version of scientific notation. Where Java displays 1.0E9, you probably expected to see 1000000000.0. This value is one billion or $10^9$. The "E9" in the output Java produces is short for "times ten raised to the power 9". The "E" stands for "exponent". In general, to interpret a number output in this form you should raise 10 to the power found after the E and multiply the number before the E by the result. The table below shows some examples of numbers written in this notation and the standard forms of the same values.

| E-notation | Standard Representation | Standard Scientific Notation |
|---|---|---|
| 1.0E8 | $100,000,000$ | $1 \times 10^8$ |
| 1.86E5 | $186,000$ | $1.86 \times 10^5$ |
| 4.9E-6 | .0000049 | $4.9 \times 10^{-6}$ |

Not only does Java display large numbers using this notation, it also recognizes numbers typed in this format as part of a program. So, if you really like scientific notation, you can include a statement like

```
avogadro = 6.022E26;
```

in a Java program.

### 7.3.2 Choosing a Numeric Type

While changing the declarations of `total` from `int` to `double` is sufficient to make our calculator work with numbers that are not integers, we could go even further. We could convert all of the `int` variables in our program into `double` variables and the program would still work as desired. In particular, we could declare the variable `digitsEntered` as a `double`. Any value that is an integer is also a rational number. Therefore, even though the values associated with `digitsEntered` will always be integers, it would not change the program's behavior if we simply told Java that this name could be associated with any number. We mention this to highlight an important question. Why does Java distinguish between the types `int` and `double`? Why isn't a single numeric type sufficient?

To understand why Java includes both of these types, consider the following two problems. First, suppose you and two friends agree to share a bottle of soda, but that at least one of you is a bit obsessive and insists that your portions have to be exactly equal. If the bottle contains 64 ounces of soda, how much soda should each of you receive?(Go ahead. Take out your calculator.)

Now, suppose that you are the instructor of a course in which 64 students are registered and you want to divide the students into 3 laboratory sections. How many students should be assigned to each section?

With a bit of luck, your answer to the first question was 21 1/3 ounces. On the other hand, even though the same numbers, 64 and 3, appear in the second problem, "21 1/3 students" would probably not be considered an acceptable answer to the second problem (at least not by the student who had to be chopped in thirds to even things out). A better answer would be that there should be two labs of 21 students and a third lab with 22 students.

The point of this example is that there are problems in which fractional results are acceptable and other problems where we know that only integers can be used. If we use a computer to help us solve such problems, we need a way to inform the computer whether we want an integer result or not.

In Java, we do this by choosing to use `int`s or `double`s. For example, if we declare three instance variables:

```
private double ounces;
private double friends;
private double sodaRation;
```

and then execute the assignment statements

```
ounces = 64;
friends = 3;
sodaRation = ounces/friends;
```

the number associated with the name `sodaRation` will be 21.33333... On the other hand, if we declare the variables

```
private int students;
private int labs;
private int labSize;
```

and then execute the assignments

```
students = 64;
labs = 3;
labSize = students/labs;
```

the number associated with `labSize` will be 21. In the first example, Java can see that we are working with numbers identified as `double`s, so when asked to do division, it gives the answer as a `double`. In the second case, since Java notices we are using `int`s, it gives us just the integer part of the quotient when asked to do the division.

Of course, the answer we obtain in the second case, 21, isn't quite what we want. If all the labs have exactly 21 students, there will be one student excluded. We would like to do something about such leftovers. Java provides an additional operator with this in mind. When you learned to divide numbers, you probably were at first taught that the result of dividing 64 by 3 was 21 with a remainder of 1. That is, you were taught that the answer to a division problem has two parts, the quotient and the remainder. When working with integers in Java, the `/` operator produces the quotient. The percent sign can be used as an operator to produce the remainder. Thus, `64/3` will yield 21 in Java and `64 % 3` will yield 1. In general, `x % y` will only equal 0 if `x` is evenly divisible by `y`.

Used in this way the percent sign is called the *mod* or *modulus* operator. We can use this operator to improve our solution to the problem of computing lab sizes by declaring an extra variable

```
private int extraStudents;
```

and adding the assignment

```
extraStudents = students % labs;
```

**Exercise 7.3.1** *For each numerical value below, decide which values could be stored as* `int`*s and which must be stored as* `double`*s.*

**a.** *the population of Seattle*

**b.** *your height in meters*

**c.** *the price of a cup of coffee in dollars*

**d.** *the inches of rain that have fallen*

**e.** *the number of coffee shops in Seattle*

**f.** *the number of salmon caught in a month*

### 7.3.3 Arithmetic with `doubles` and `ints`

The distinction between `double`s and `int`s in Java is a feature intended to allow the programmer to control the ways in which arithmetic computations are performed. In Section 5.1.1, we saw that the way in which Java interprets a plus sign differs significantly based on whether the operands to the operator are `String`s or numbers. In one case, `+` is interpreted as concatenation. In the other, `+` is interpreted as addition. In a similar way, Java's interpretation of all its arithmetic operators depends on the numeric types to which the operands belong. If all of the operands to an arithmetic operator are `int` values, the operation will return an `int` value. If any of the operands are `double`s, then the operation will return a `double` as its result.

The consequences of this distinction are most pronounced when division is involved. When we use a numeric literal in a program, Java assumes the value should be interpreted as an `int` if it contains no decimal point and as a `double` if it contains a decimal point (even if the fractional part is 0). As a result, the expression

```
1/2
```

produces the `int` value 0 as its result while the expressions

```
1.0/2
```

and

```
1/2.0
```

and

```
1.0/2.0
```

all produce the `double` 0.5.

Because of this, the order in which Java performs arithmetic operations sometimes becomes critical in determining the value produced. This order is determined by two basic *precedence rules*:

- Division and multiplication are considered to be of higher precedence than addition and subtraction. That is, unless constrained by the use of parentheses in an expression, Java will perform multiplications and divisions before additions and subtractions.
- Operations of equal precedence are performed in order from left to right.

These rules are not unique to Java. We follow the same rules when interpreting formulas in mathematics. That is why we interpret the formula

$$3x + 1$$

as equivalent to $(3x) + 1$ rather than $3(x + 1)$. We know that we are supposed to perform the multiplication before the addition. Because of issues like the interactions between `int`s and `double`s, however, these rules often become more critical in Java programs than they typically are in math classes.

For example, in Java, the expressions

```
64.0*1/3
```

and

```
1/3*64.0
```

produce different values! In the first expression, the computer first multiples 64.0 times 1. Since the first operand is a `double`, the computer will produce the `double` result 64.0 and then divide this by the `int` 3. Again, since at least one operand of the division is `double` the computer will produce the `double` result 21.3333...

On the other hand, when the second expression is evaluated, the process begins with the division of 1 by 3. In this case, both operands are `int`s so the result produced must be an `int`. In particular, the result of this division will be 0. This result is then multiplied by the `double` 64.0. The final result will be the `double` value 0.0.

While Java distinguishes between `int`s and `double`s, it recognizes that they are related. In particular, in contexts where one should technically have to provide a `double`, Java will allow you to use an `int`. This was already illustrated in the example above where we declared an instance variable

```
private double ounces;
```

and then wrote the assignment

```
ounces = 64;
```

The literal 64 is identified by Java as an `int` because it contains no decimal point. The variable is declared to be a `double`. Normally, Java considers an assignment invalid if the type of the value described by the expression to the right of the equal sign is different from the type of the variable on the left side of the equal sign. The assignments

```
ounces = new JTextField( 10 );
```

and

```
ounces = "10";
```

would be considered illegal because `ounces` is a `double` rather than a `JTextField` or a `String`. Java will, however, accept the assignment

```
ounces = 64;
```

even though it assigns an `int` to a variable that is supposed to refer to a `double`.

Java is willing to convert `int`s into `double`s because it knows there is only one reasonable way to do the conversion. It simply adds a ".0" to the end of the `int`. On the other hand, Java knows that there are several ways to convert a `double` into an `int`. It could drop the fractional part or it could round. Because it can not tell the correct technique to use without understanding the context or purpose of the program, Java refuses to convert a `double` into an `int` unless explictly told how to do so using mechanisms that we will discuss later. Therefore, given the instance variable declaration

```
private int students;
```

Java would reject the assignment

```
students = 21.333;
```

as erroneous. It would also reject the assignment

```
students = 64.0;
```

Even if the only digit that appears after the decimal point in a numeric literal is 0, the presence of the decimal point still makes Java think of the number as a `double`.

**Exercise 7.3.2** *What is the value of each of the following expresions?*

**a.** `12 / 5`

**b.** `35 / 7`

**c.** `15.0 / 2.0;`

**d.** `65.0 / 4;`

**e.** `79 % 12;`

**Exercise 7.3.3** *Given that the variable* `ratio` *is of type* `double`*, what value will be associated with the* `ratio` *by each of the following assignments?*

**a.** `ratio = 12.0 / 4;`

**b.** `ratio = 32 / 6;`

**c.** `ratio = 48.0 / 5;`

**d.** `ratio = 22 % 8;`

### 7.3.4 Why are Rational Numbers Called `double`?

As a final topic in this section, we feel obliged to try to explain why Java chooses to call numbers that are not integers `double`s rather than something like `real` or `rational`. The explanation involves a bit of history and electronics.

To allow us to manipulate numbers in a program, a computer's hardware must encode the numbers we use in some electronic device. In fact, for each digit of a number there must be a tiny memory device to hold it. Each of these tiny memory devices costs some money and, not long ago, they cost quite a bit more than they do now. So, if a program is only working with small numbers, the programmer can reduce the hardware cost by telling the computer to set aside a small number of memory devices for each number. On the other hand, when working with larger numbers, more memory devices should be used.

On many machines, the programmer is not free to pick any number of memory devices per number. Instead only two options are available: the "standard" one, and another that provides **double** the number of memory devices per number. The name of the Java type `double` derives from such machines.

While the cost of memory has decreased to the point where we rarely need to worry that using `double`s might increase the amount of hardware memory our program uses, the name does reflect an important aspect of computer arithmetic. Since each number represented in a computer is stored in physical devices, the total number of digits stored is always limited.

The number of binary digits used to represent a number limits the range of numbers that can be stored. In the computer's memory, thirty-one binary digits are used to store the numeric value of each `int`. As a result, as we mentioned in Section 5.4, the values that can be processed by Java as `int`s range from -2,147,483,648 ( $= -2^{31}$) to 2,147,483,647 ( $= 2^{31} - 1$ ). If you try to assign a number outside this range to an `int` variable, Java is likely to throw away some of the digits, yielding an incorrect result. If you need to write a program that works with very large integers, there is another type that is limited to integer values but that can handle numbers with twice as many digits. This type is called `long`. There is also a type named `short` that uses half as much memory as `int` to represent a number. For numbers that are not integers, there is also a type named `float` that is like `double` but uses half as many bits to represent numeric values. All of these types are primitive types.

The range of numbers that can be stored as `double` values is significantly larger than even the `long` type. The largest `double` value is approximately $1.8 \times 10^{308}$ and the smallest `double` is approximately $-1.8 \times 10^{308}$. This is because Java stores `double` values as you might write numbers in scientific notation. It rewrites each number as a value, the mantissa, times 10 raised to an appropriate exponent. For example, the number

$$32,953,923,804,836,184,926,273,582,140,929.584,289$$

might be written in scientific notation as

$$3.295392380483618492627358214092958428 9 \times 10^{31}$$

Java, however, does not always encode all the digits of the mantissa of a number stored as a `double`. The amount of memory used to store a `double` enables Java to record approximately 15 significant digits of each number. Thus, Java might actually record the number used as an example above as

$$3.295392380483618 \times 10^{31}$$

This means that if you use numbers with long or repeating sequences of digits, Java will actually be working with approximations of the numbers you specified. The results produced will therefore be slightly inaccurate. Luckily, for most purposes, the accuracy provided by 15 digits of precision is sufficient.

There is one final aspect of the range of `double` values that is limited. First, the smallest number greater than 0 that can be represented as a `double` is approximately $5 \times 10^{-324}$. Similarly, the largest number less than 0 that can be represented as a `double` is approximately $-5 \times 10^{-324}$.

## 7.4 `boolean` Variables

Now that our calculator works with `double`s, it would certainly make sense to add a decimal point key to its interface. In this section, we will consider the changes required to make this addition. The interesting part of this process will be making sure that the decimal point key is used appropriately. In particular, we have to make sure that it is not possible to enter a number that contains more than one decimal point. Implementing this restriction as part of our `buttonClicked` method will give us the opportunity to explore one remaining aspect of the type `boolean`, the use of variables that are associated with `boolean` values.

Adding a decimal point key to our program's interface would be quite simple if we were willing to assume that the user would always use it correctly. We would add a declaration of the form

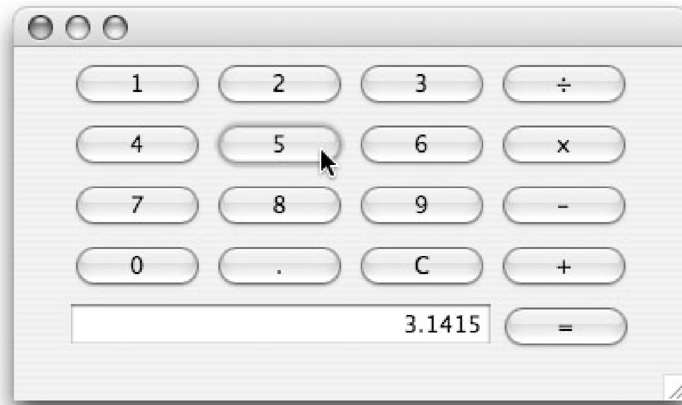


Figure 7.8: Layout for calculator interface with decimal point key

```
// Button used to enter decimal points
private JButton decimalPoint = new JButton( "." );
```

to the instance variables already declared in our class. We would also have to add this new button to the content pane. Of course, we would first have to decide how to rearrange the existing buttons to make room for this addition while still presenting an interface that is easy to use. With a bit of shuffling, we came up with the layout shown in Figure 7.8. This would require reordering the instructions in our program's constructor that add GUI components to the content pane.

Finally, since it is now possible for a user to enter a number that contains a decimal point, using the `Integer.parseInt` method to convert the number from the `String` type to the `double` type is no longer appropriate. Luckily, there is a very similar method named `Double.parseDouble` that is designed to convert `String`s that look like `double` literals into `double` values. Therefore, we would replace the initialized local variable declaration

```
int numberEntered = Integer.parseInt( entry.getText() );
```

found in our `buttonClicked` method with the declaration

```
double numberEntered = Double.parseDouble( entry.getText() );
```

Like the `Integer.parseInt` method, the `Double.parseDouble` method will cause a program error if the `String` that is passed to it as an argument does not take the expected form. Our program uses the contents of the `entry` text field as the argument to `Double.parseDouble`. Therefore, the program should make sure that the contents of this field will always look like a valid `double` literal before we try to convert it to a `double`. If we made no additional changes to our program, this would not be the case. It would be possible for a user to press the decimal key several times while entering a number. To ensure that this is not possible, we must modify our `buttonClicked` method to keep track of how many times the decimal point key is pressed in such a way that it can ignore attempts to place several decimal points in one number just as it already ignores attempts to enter a number with more than 9 digits.

We could accomplish this by using an `int` variable to count how many decimal points have been entered, much as we used the variable `digitsEntered` to count how many digit keys have been

pressed. If we did this, however, the variable we used would never take on any value other than 0 or 1. We don't really need to know how many decimal points have been entered, we simply need to know whether a decimal point has been entered or not. A variable like `digitsEntered` provides the ability to answer the question "How many digits have been entered?" To handle decimal points correctly, all we need is to be able to answer the question "Has a decimal point been entered yet?" This is the type of question we use when writing the condition of an `if` statement. It is a question with only two possible answers. The type `boolean` provides two values, `true` and `false`, that can be used to represent the answer to such a question. Therefore, we can use a `boolean` variable rather than an `int` variable to record the information we need to remember about the decimal points that have been entered.

To do this we will include an instance variable declaration of the form

```
// Remember whether or not a decimal point has been entered
private boolean decimalPtEntered = false;
```

We will also have to add assignments that will associate this variable with the correct value at appropriate places in our program. When the user does press the decimal point key, we need to make sure our program executes an assignment of the form

```
decimalPtEntered = true;
```

When the user presses an operator key or the clear button, we will have to execute an assignment of the form

```
decimalPtEntered = false;
```

Then, it will be possible to ignore duplicate decimal points by using an `if` statement of the form

```
if ( ! decimalPtEntered ) {
    // Add a decimal point to the number being entered
    ...
}
```

This `if` statement's condition looks different from all the other examples of `if` statements we have considered. It does not contain any of the relational operators like `==` or `<`. This shows that it is not necessary to include such an operator in a condition. All that Java requires is that the condition we provide in an `if` statement describes a `boolean` value. The expression `!decimalPtEntered` describes the value obtained by reversing the `true/false` value currently associated with `decimalPtEntered`. If the value associated with this variable answers the question "Has a decimal point been entered yet?" then by using the `!` operator to reverse its value we are telling Java that it should only execute the body of the `if` statement if no decimal point has been entered.

All of the changes required to handle the decimal point key correctly in the `buttonClicked` method are shown in Figure 7.9. The code is slightly abridged to fit on one page. We have replaced the 5-way if statement that determines which operator to apply with a single comment.

In addition to the changes discussed above, it is worth noting that the `boolean` variable `decimalPtEntered` is also used in the condition of the `if` statement that determines when the `entry` text field is cleared. Here it is used in a more complex condition involving the logical operator `&&` since clearing the field depends on whether either a digit or decimal point has been entered.

```
// Accept digits entered and perform arithmetic operations requested
public void buttonClicked( JButton clickedButton ) {
        if ( clickedButton == minus  || clickedButton == times ||
             clickedButton == divide || clickedButton == plus  ||
             clickedButton == equals ) {
           // Apply last operator to the number entered and current total
           if ( digitsEntered > 0 ) {

                double numberEntered = Double.parseDouble( entry.getText() );

                . . .  // 5-way if statement omitted to compress code

                entry.setText( "" + total );
                digitsEntered = 0;
                decimalPtEntered = false;

                lastOperator = clickedButton;
           }

        } else if ( clickedButton == clear ) {
            //  Zero the total and clear the entry field
            total = 0;
            digitsEntered = 0;
            decimalPtEntered = false;
            entry.setText( "0" );
            lastOperator = equals;

        } else {  // If clickedButton is a numeric or decimal key, add to entry
            if ( digitsEntered == 0 && ! decimalPtEntered) {
                entry.setText( "" );
            }

            if ( clickedButton == decimalPoint ) {
                if ( ! decimalPtEntered ) {
                    decimalPtEntered = true;
                    entry.setText( entry.getText() + "." );
                }
            } else {
                digitsEntered = digitsEntered + 1;
                entry.setText( entry.getText() + clickedButton.getText() );
            }
        }
    }
```

Figure 7.9: Handling decimal points using a `boolean` variable

## 7.5 Summary

In this chapter, we explored the basic mechanisms Java provides for working with numeric data in a program. Java distinguishes its numeric types from types such as the GUI components we have used in many of our examples in several ways. Numeric values belong to what are called the set of primitive types, while GUI components are examples of members of object types or classes. Values of primitive types cannot be constructed. Instead, they can be described using literals in our programs. In addition, operators rather than methods are provided to manipulate numeric types.

Java provides several distinct types for representing numbers. The biggest difference between these types is that while the types `double` and `float` are designed for working with numbers with fractional parts, Java's other numeric types, including `long`, `int`, and `short` can only represent whole numbers. The distinctions between `double`s and `float`s and between `long`s, `int`s, and `short`s involve the range of values that are supported by these types.

Java provides the operators `+`, `-`, `*`, and `/` for the standard operations of addition, subtraction, multiplication and division. These operators can be used with any of the numeric types. When all of the operands to one of these operators are integers, the result will be an integer. If any of the operands is a `double`, the result will be a `double`. For the integer types, Java also provides the `%` operator which returns the remainder associated with a division operation.

In addition to its numeric types, Java provides one additional type that is considered a primitive type, the type `boolean`. Java provides the logical operators `&&`, `||`, and `!` for manipulating `boolean` values. It also provides relational operators that produce `boolean` values as their results. Java requires that the conditions used in `if` statements be expressions that produce `boolean` values as their results.