Chapter 9

Doomed to Repeat

Looking up synonyms for the word "repetitive" in a thesaurus yields a list that includes the words "monotonous", "tedious", "boring", "humdrum", "mundane", "dreary", and "tiresome". This reflects the fact that in life, repeating almost any activity often enough makes it unpleasant. In computer programs, on the other hand, repetition turns out to be very exciting. In some sense, it is the key to exploiting the power of the computer.

To appreciate this, think about how quickly computers can complete the instructions we place in a program. The speed of a modern computer is measured in gigahertz. That means that such a computer can perform billions of steps per second. These are very small steps. It may take tens or hundreds of them to complete a single line from a Java program. Even so, your computer is probably capable of completing the instructions in millions of lines of Java instructions in seconds. Any programmer, on the other hand, is unlikely to be fast enough to write a million lines of Java instructions in an entire lifetime. Basically, a computer can execute more instructions than we can write for it. This means the only possible way to keep a computer busy is to have it execute many of the instructions we write over and over and over again.

Programming languages provide constructs called *loops* to enable us to specify that a sequence of instructions be executed repetitively. In this chapter, we will introduce one of Java's looping constructs, which is called the *while loop*.

To use `while` loops or any other looping construct effectively, one must learn to write instructions that perform useful work when executed repeatedly. Typically, this means that even though the computer will execute exactly the same instructions over and over again, these instructions should do something at least slightly different each time they are executed. We will see that this can be accomplished by writing instructions that depend on variable values or other aspects of the computer's state that are modified during their execution. In particular, we will introduce several important patterns for writing loops including counting loops and input loops.

9.1 Repetition without Loops

Before talking about new Java constructs, we should explore the idea that executing the same sequence of instructions over and over again does not necessarily mean doing exactly the same thing over and over again. You have already seen examples of this in many of the programs we have examined. As a very simple case, consider the numeric keypad program presented in Figure 3.12. The display produced by this program is shown in Figure 9.1.

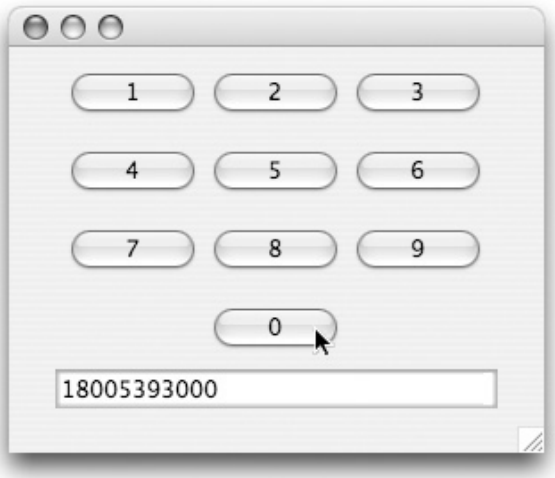

Figure 9.1: A numeric keypad user interface

The `buttonClicked` method in the program contains just one line of code:

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

The name associated with the text field displayed at the bottom of the program's window is `entry`, and `clickedButton` is the formal parameter name associated with the button whose clicking caused the execution of the `buttonClicked` method. As a result, each time a button is clicked, its digit value is appended to the string of digits shown in the text field.

In some sense, every time any button in the keypad is pressed, this program does exactly the same thing. It executes the single line that forms the body of the `buttonClicked` method. The effect of executing this line, however, depends both on which button was just pressed and on what buttons were previously pressed. Therefore, the behavior that results each time the line is executed is slightly different.

If the first button pressed is the "1" key, then executing this line will cause a "1" to be displayed in the text field named `entry`. If the user next presses the "2" key, the same line will be executed, but instead of causing "1" to be displayed, the pair "12" will be displayed. Even if the user had pressed exactly the same key both times, the first execution would display "1" while the second would display "11".

The key to the way in which this code's behavior varies is the fact that it both depends upon and changes the contents of the text field `entry`. It accesses the contents of this field using the `getText` method and then changes it using `setText`.

There are many other ways in which we can write code that bears repetition by making the code both depend upon and change some aspect of the state of the computer. One very common approach is to write code that depends on and changes the values associated with one or more variables. To illustrate this, we will consider a rather impractical way to construct the keyboard shown in Figure 9.1.

The constructor used to create the keypad which was originally presented in Figure 3.12 is repeated for convenience in Figure 9.2. Ten lines of this code are extremely similar to one another, but they are not quite identical. We are looking for examples where executing exactly the same

```
   public NumericKeypad() {
      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( DISPLAY_WIDTH );
      contentPane.add( entry );
}
```

Figure 9.2: Constructor that creates a keyboard interface

instructions over and over again produces useful results, so lets think about how we can replace these nearly identical instruction with instruction that are actually identical.

The only difference between the lines that invoke `contentPane.add` is the value that is provided as a label for the `JButton`s being constructed. We can eliminate this difference by replacing the literals `"1"`, `"2"`, etc. with an expression that depends on a variable.

Recall that if `buttonNumber` is an `int` variable associated with the value 1, then the expression

```
"" + buttomNumber
```

produces the `String` value

```
"1"
```

Therefore, if the value associated with the variable `buttonNumber` is 1, then the statement

```
contentPane.add( new JButton( "" + buttonNumber ) );
```

is equivalent to

```
contentPane.add( new JButton( "1" ) );
```

Similarly, if `buttonNumber` is associated with 2, then the same statement is equivalent to

```
contentPane.add( new JButton( "2" ) );
```

Given these observations, it should be clear that if we initially associate the value 1 with `buttonNumber` then we could replace the first nine lines of the code shown in Figure 9.2 with nine copies of the two instructions

```
contentPane.add( new JButton( "" + buttonNumber ) );
buttonNumber = buttonNumber + 1;
```

The result of executing this pair of instructions depends on the value of the variable `buttonNumber`, but the instructions also change the value of this variable. As a result, executing one copy of this pair of instructions will produce a button with the label "1", executing a second copy will produce a button displaying "2", and so on. Thus, nine copies of these two lines can be used to produce the buttons "1" through "9".

If we tried to use a tenth copy of these two lines to produce the last button, we would be disappointed. Executing a tenth copy of these two lines would produce a button labeled with "10". Unfortunately, the last button in the keyboard should be labeled with "0" rather than "10". It is possible, however, to write two instructions that will correctly generate all ten buttons if executed ten times.

In Section 7.3.2, we introduced the modulus operator, `%`. If `a` and `b` are integer-valued expressions, then `a % b` produces the remainder that results when `a` is divided by `b`. If we divide any number between 1 and 9 by 10, then the quotient is 0 and the remainder is just the original number. If we divide 10 by 10, however, the quotient is 1 and the remainder is 0. Therefore, the expression `buttonNumber % 10` will return the value of `buttonNumber` if that value falls between 1 and 9 and 0 if the value of `buttonNumber` is 10.

We can, therefore, use ten copies of the two instructions

```
contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
buttonNumber = buttonNumber + 1;
```

to create the entire keypad as shown in Figure 9.3. The last copy of the statement

```
buttonNumber = buttonNumber + 1;
```

is clearly unnecessary, but it accomplishes our goal of replacing the 10 similar instructions in the original code with 10 identical code segments.

9.2 Worth Your While

You may have the feeling that we didn't accomplish anything useful in the preceding section. Replacing 10 somewhat repetitive lines of code with 20 even more repetitive lines is not usually considered a good thing. The changes we have made, however, put us in position to use a new Java language construct called the *`while` statement* or *`while` loop* to make the code required to construct the keyboard much more compact.

The general form of a `while` statement is

```
while ( condition )
    statement
```

```
  public NumericKeypad() {
     this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

     int buttonNumber = 1;

     contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
     buttonNumber = buttonNumber + 1;
     contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
     buttonNumber = buttonNumber + 1;
     contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
     buttonNumber = buttonNumber + 1;
     contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
     buttonNumber = buttonNumber + 1;
     contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
     buttonNumber = buttonNumber + 1;
     contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
     buttonNumber = buttonNumber + 1;
     contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
     buttonNumber = buttonNumber + 1;
     contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
     buttonNumber = buttonNumber + 1;
     contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
     buttonNumber = buttonNumber + 1;
     contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
     buttonNumber = buttonNumber + 1;

     entry = new JTextField( DISPLAY_WIDTH );
     contentPane.add( entry );
}
```

Figure 9.3: Extremely repetitive code to create a keyboard interface

```
    public NumericKeypad() {
       this.createWindow( WINDOW_WIDTH, WINDOW_HEIGHT );

       int buttonNumber = 1;

       while ( buttonNumber <= 10 ) {
           contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
           buttonNumber = buttonNumber + 1;
       }

       entry = new JTextField( DISPLAY_WIDTH );
       contentPane.add( entry );
}
```

Figure 9.4: Using a `while` statement to construct a key pad

The keyword `while` and the condition included are called the *loop header* and the statement included is called the *loop body*. It is very common to use a compound statement formed by placing a series of statements in curly braces as the body of a `while` loop. That is, in practice, `while` statements are almost always written as

```
while ( condition ) {
     one or more statements
}
```

The condition included in the header of a `while` statement must be an expression that returns a value of the type `boolean`. When a program's execution reaches a `while` statement, the computer repeatedly executes the statement(s) found in the body of the loop until evaluating the condition yields `false`.

To see how to use this construct, recall the code we devised for creating a keypad in the last section. To draw the keypad, all we did was repeatedly execute the two statements

```
contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
buttonNumber = buttonNumber + 1;
```

by placing 10 copies of these statements in our code. Now, we can replace these 20 lines with the four line `while` loop shown in Figure 9.4.

Let us consider how the loop in Figure 9.4 would be executed by a computer. Just before beginning to execute the while loop, the computer will process the declaration

```
int buttonNumber = 1;
```

which initializes `buttonNumber` to refer to the value 1. Next, the computer will begin the process of executing the loop by checking to see if the condition `buttonNumber <= 10` specified in the loop header is `true`. Given that the value 1 has just been associated with `buttonNumber`, the condition will be `true` at this point, and the computer will proceed to execute the two statements in the loop body. The first of these statements will place the "1" button in the program's window. The second statement will change the value associated with `buttonNumber` from 1 to 2.

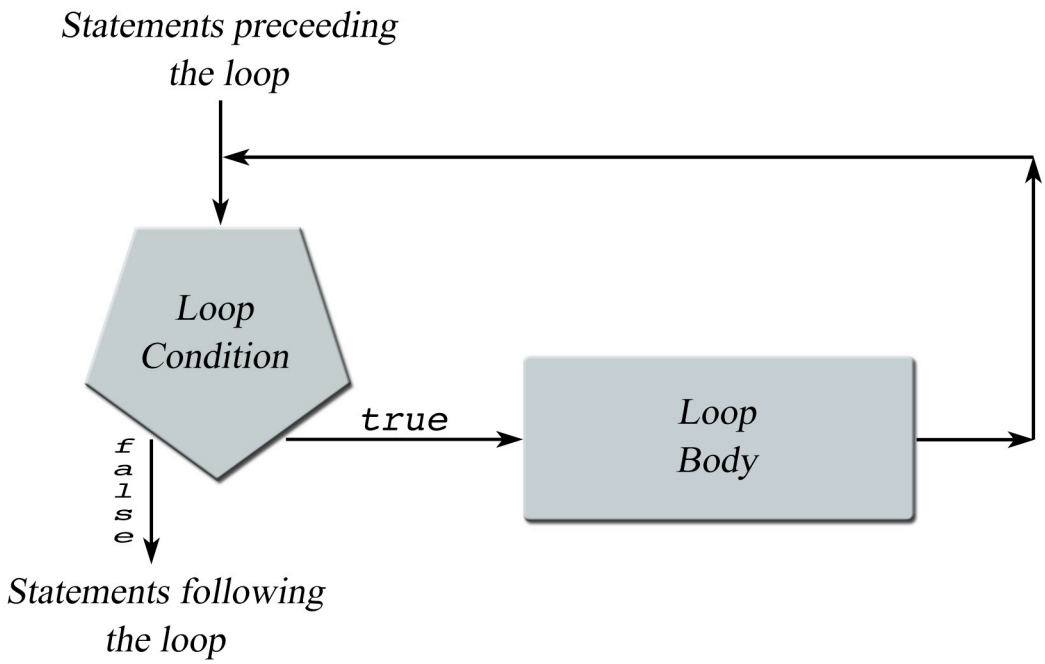

Figure 9.5: Flow of execution through a `while` loop

After the first execution of the loop body, the computer will again check whether evaluating the condition of the loop produces `true` or `false`. Executing the body will have increased the value associated with `buttonNumber` to 2, but it will still be true that `buttonNumber <= 10`. Accordingly, the computer will proceed to execute the body a second time, placing the "2" button in the window and changing the value associated with `buttonNumber` to 3.

The process will be repeated in very much the same way 7 more times for the buttons "3" through "9". After the ninth execution of the loop body, the computer will again verify that the condition `buttonNumber <= 10` is `true`. The value 10 will be associated with `buttonNumber`, so it will proceed to execute the two statements in the loop body a tenth time.

This last execution of the loop places the "0" button on the screen and associates `buttonNumber` with the value 11. At this point, the condition `buttonNumber <= 10` is no longer `true`. Therefore, after the tenth execution of the loop body, the computer will check the condition for the eleventh time and notice that the condition is now `false`. This will cause the computer to realize that the execution of the loop is complete. The machine will proceed to execute the two statements that follow the loop in the constructor:

```
entry = new JTextField( DISPLAY_WIDTH );
contentPane.add( entry );
```

The diagram in Figure 9.5 illustrates the way in which Java executes all `while` loops.

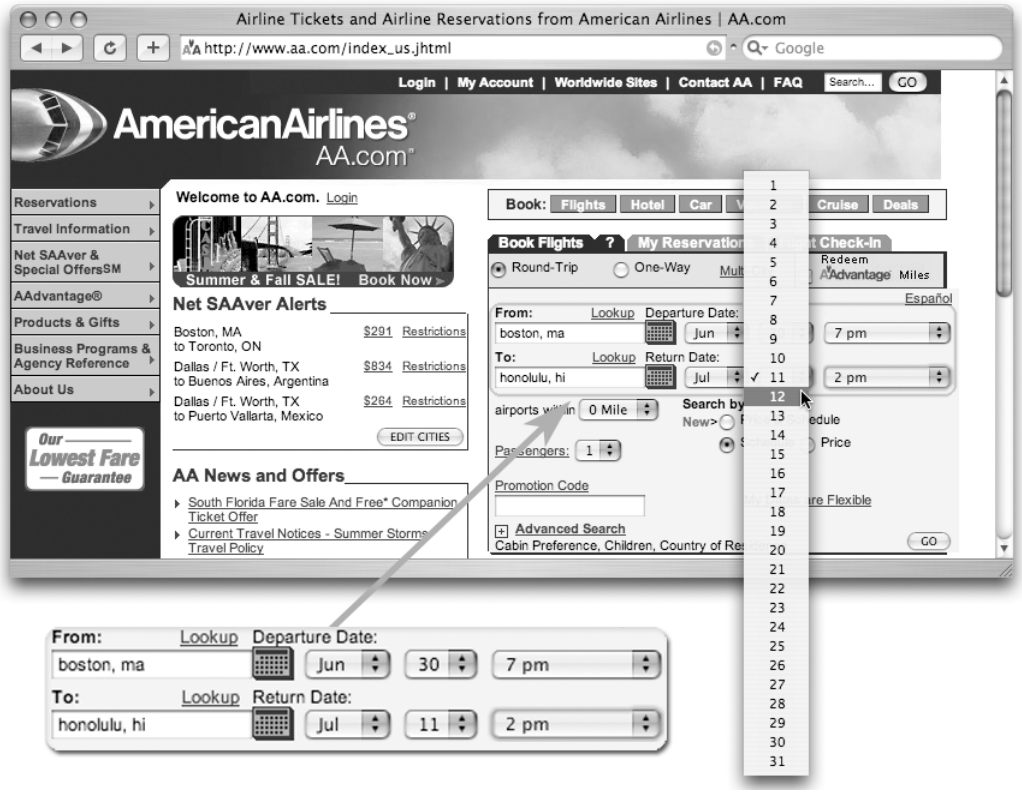

Figure 9.6: American Airlines on-line reservation interface

9.3 Count to Ten

The loop we presented in the previous section is an example of a very common and important type of loop: a loop that counts. In that loop, we counted how many buttons had been displayed to determine both how to label each button and how many times to execute the body of the loop. To solidify your understanding of the basics of `while` loops, we will consider several additional examples of counting loops before moving on to explore other ways to write loops.

In our first example, we used a counting loop to place buttons in a window. In this section, we will use very similar loops to place items in pop-up menus. As motivation, imagine that you decided that a little time in a tropical paradise would be nice and set out to plan a trip to Hawaii. Among other arrangements, you would have to book a flight. Chances are that you would look for the best deal you could find on airline websites and travel sites like Orbitz, Kayak, and Travelocity.

On most of these sites, you would encounter an interface similar to that shown in Figure 9.6. This figure shows the contents of the American Airlines reservation web site captured as a user is changing the date for the return flight from July 11th to July 12th. The pop-up menu obscures several of the component displayed beneath it. We have included a slightly enlarged image of these components below the image of the browser window. We will explore how to write Java code to create components similar to these in a Java program rather than in a web page.

The interface we have in mind for our Java program is shown in Figure 9.7. This figure includes two images of the program's window, one showing all of the components in their idle state and the

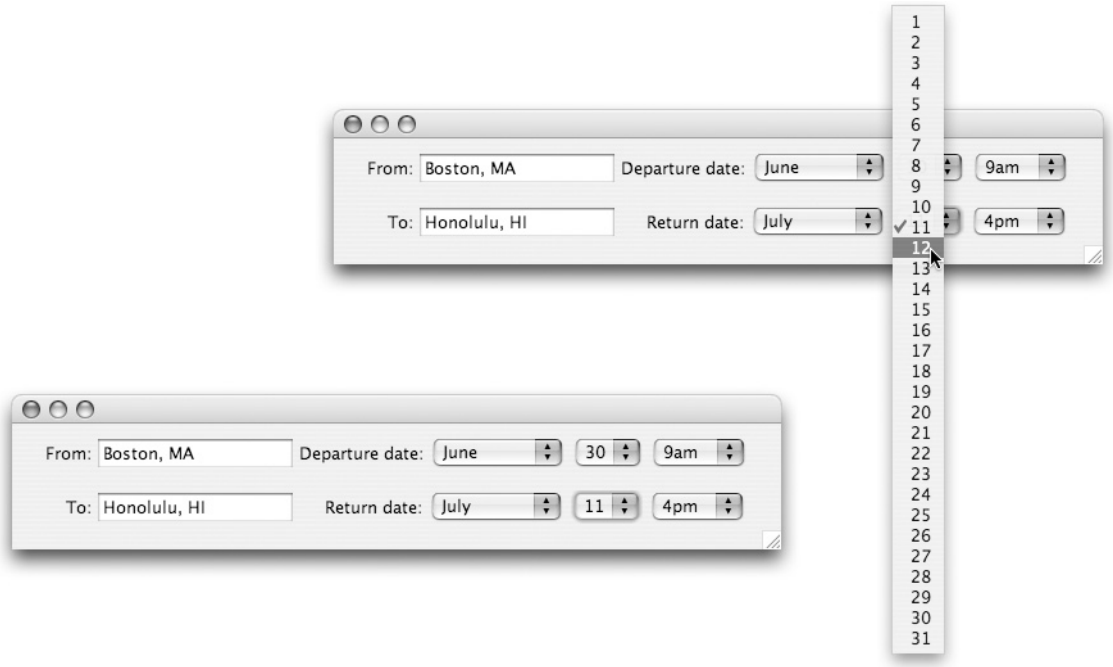

Figure 9.7: A Java version of an airline reservation interface

```
int buttonNumber = 1;

while ( buttonNumber <= 10 ) {
    contentPane.add( new JButton( "" + ( buttonNumber % 10 ) ) );
    buttonNumber = buttonNumber + 1;
}
```

Figure 9.8: A counting loop to create keyboard buttons

other showing one of the menus being used to select a return date.

The interface consists of two, nearly identical rows of components. The top row is intended for specifying the departure location, date, and time. The bottom row is for the destination and return flight date and time. Each row contains a `JTextField` for the departure or destination city, a few `JLabel`s, and three `JComboBox`es. We will focus on the last two `JComboBox`es on each row.

Our first goal is to write a loop to fill the menu used to select a date within a month. This is the menu shown in use in Figure 9.7. The loop we need is almost identical to the loop we used to create the buttons for our keyboard in Figure 9.4, so we start by considering the exact roles of the components of that code. The lines of interest are repeated in Figure 9.8.

The variable `buttonNumber` was used to keep track of how many buttons had been added to the content pane. We need to keep track of how many dates have been added to the menu we are creating. Accordingly, the first line of code we need is something like

```
int date = 1;
```

```
// Fill a menu with the numbers of the days in a particular month
private void fillDateMenu( JComboBox dateMenu, int lengthOfMonth ) {

    dateMenu.removeAllItems();

    int date = 1;
    while ( date <= lengthOfMonth ) {
        dateMenu.addItem( "" + date );
        date = date + 1;
    }
}
```

Figure 9.9: A method that uses a counting loop to fill a menu with dates

The condition in the loop for keyboard buttons stopped the loop as soon as 10 buttons had been created. We want our new loop to stop as soon as an entry has been added for every day of the month. Unfortunately, different months have different numbers of days. So, we assume that our loop will be preceded by code to associate the appropriate value with a variable named `lengthOfMonth`. Then, we can use a loop header of the form

```
while ( date <= lengthOfMonth ) {
```

Finally, the body of the loop should contain one statement to add an item to the menu each time the loop executes and one statement to increase the `date` variable's value by 1. Assuming the menu is named `dateMenu`, the two instructions

```
dateMenu.addItem( "" + date );
date = date + 1;
```

could be used as the body of the loop.[1]

We can put all these pieces together to form a `private` method that fills an existing `JComboBox` with the list of dates appropriate for a particular month. Figure 9.9 shows the complete code for such a method. The method is designed so that it can be invoked to update the list of dates included in the menu whenever the user selects a new month from the associated menu of month names. It therefore begins by removing any existing entries from the menu and then fills the menu using the counting loop we have described.

It should be clear that the loops in Figures 9.8 and 9.9 are extremely similar. They both have the basic form illustrated by the following mix of Java and English:

```
Associate starting value with a counter variable
while ( counter is no greater than final value desired ) {
      Do a bit of interesting work using the counter
      Increase the value associated with the counter
 }
```

[1]This code can be simplified if you are using Java 5.0 or later. The `addItem` method can now accept numeric parameter values. Therefore, you can replace the parameter `"" + date` with just `date`.

This, in fact, is the basic form of all counting loops. There are, however, many ways to vary this basic pattern.

We can reverse the order of the two components of the loop body as long as we adjust the starting value and the loop condition appropriately. That is, the following loop could be used in place of the loop used in Figure 9.9:

```
int date = 0;
while ( date < lengthOfMonth ) {
    date = date + 1;
    dateMenu.addItem( "" + date );
}
```

This loop reveals one slightly subtle point about the way in which the condition controls the execution of the loop. During the final repetition of this loop, the computer first executes the instruction

```
date = date + 1;
```

At this point, the condition `date < lengthOfMonth` will no longer be `true`. The computer, however, does not immediately stop executing the body of the loop. The condition that controls the execution of a loop is checked exactly once for each repetition of the loop body *before* the computer begins to execute the statements in the loop body. Once the computer begins to execute the loop body, it does not check the condition again until the process of executing the body is complete. Therefore, even though `date` becomes equal to `lengthOfMonth` in the middle of the last execution of the loop body, the computer will proceed to execute the statement

```
dateMenu.addItem( "" + date );
```

placing the last entry in the window.

We don't always have to count by ones in our loops. Just like the cheer "2, 4, 6, 8, who do we appreciate", if we replace the 1 in

```
date = date + 1;
```

with some other value we can count using a different increment. This would not be a sensible change in this loop because we want the menu to contain every number from 1 up to the last day in the month. But there are many examples where increments other than 1 are useful.

Consider the menus in our program used to indicate the desired departure times for flights. We would like to fill these menus with one entry for each of the 24 hours in a day. We can (and shortly will) fill this menu with entries including the suffixes "AM" and "PM" to distinguish between morning and afternoon departure times. An alternative that avoids these suffixes is to use "military" or "railroad" time in which the hours are numbered from 1 through 24. In this system, 7AM becomes 700 or "seven hundred hours" while 7PM is 1900 or "nineteen hundred hours." Figure 9.10 shows a menu of times built using this system. We can fill such a menu by writing a loop that counts by 100 instead of 1. The code required is shown in Figure 9.11.

We can even count backwards by decreasing the value associated with the counter variable each time the loop executes. Of course, we have to associate the counter variable with the largest value just before the loop and change the condition to test whether we have reached the smallest value.

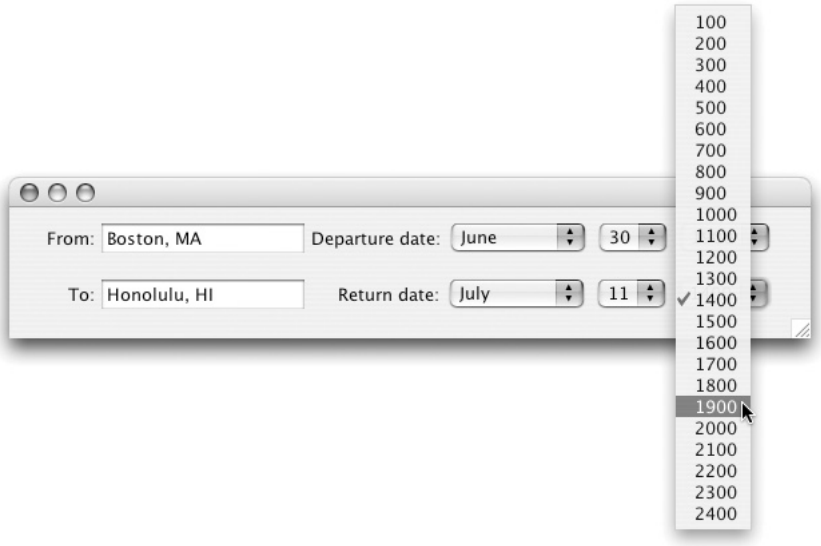

Figure 9.10: A menu of hours using 24-hour clock times

```
JComboBox timeMenu = new JComboBox();

int hour = 100;
while ( hour <= 2400) {
    timeMenu.addItem( "" + hour );
    hour = hour + 100;
}
```

Figure 9.11: A counting loop to create a 24-hour clock menu

```
JComboBox timeMenu = new JComboBox();

int hour = 2400;
while ( hour >= 100 ) {
    timeMenu.addItem( "" + hour );
    hour = hour - 100;
}
```

Figure 9.12: A counting loop to create a reversed 24-hour clock menu

Thus, if we want our 24-hour clock menu to appear with 2400 at the top and 100 at the bottom we would use the loop shown in Figure 9.12.

While counting by 100s or 10s or even backwards is useful in many situations, the most common form of counting loop is the loop that counts by 1s. In fact, Java includes a special shorthand for writing an instruction that increments a counter variable by 1. If `counter` is a numeric variable, then the statements

```
++counter;
```

and

```
counter++;
```

are both[2] equivalent to the statement

```
counter = counter + 1;
```

Thus, we could replace the loop to fill our date menu with the loop

```
int date = 0;
while ( date < lengthOfMonth ) {
    ++date;
    dateMenu.addItem( "" + date );
}
```

Similarly, the statements

```
--counter;
```

and

```
counter--;
```

decrease the value associated with a numeric variable by 1.

9.4 Nesting Instinct

In our generic template for a counting loop, one of the two parts of the loop body was described as "*Do a bit of interesting work using the counter.*" More interesting variations of counting loops can be formed by doing more interesting work. In particular, there is nothing that limits us to using just a single Java statement to describe this work. We can use as many statements as we need and we can use complex statements including `if` statements and even additional `while` loops.

To illustrate the use of a loop within a loop, consider making some additional travel arrangements. While the airlines make the reasonable assumption that no one should specify their desired flight time with more precision than the nearest hour, some car rental companies expect customers to be even more precise. For example, the Budget Rent-A-Car reservation web site includes a menu with times at 15 minute intervals. A small fragment of this menu is shown in Figure 9.13

[2]There is no difference between `++counter` and `counter++` when they are used as statements. Java, however, allows a programmer to use these constructs as expressions. Using these constructs *as expressions* can produce code that is difficult to understand. We urge you to avoid such use. When these constructs are used as expressions, they both increment `counter` and produce a value. The expression `++counter` produces the value associated with `counter` after the increment is performed. The expression `counter++` produces the value associated with `counter` before the increment.

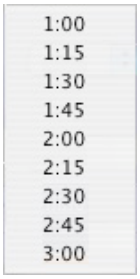

Figure 9.13: A section of the Budget Rent-A-Car pickup time menu

```
JComboBox timeMenu = new JComboBox();

int hour = 100;
while ( hour <= 2400) {
    Do a bit of interesting work using the counter.
    hour = hour + 100;
}
```

Figure 9.14: An incomplete counting loop to create a clock menu

At first, it might seem easy to adapt the code shown in Figure 9.11 to build such a menu. That code counted by 100s. What if we just changed it to count by 15s?

Unfortunately, the computer has no way to know that the value stored in the variable `hour` is supposed to represent a time of day. It only knows that it is a number. If the value of `hour` starts at 100 and we repeatedly add 15, the values placed in the menu will be 100, 115, 130, 145, 160, 175, 190, 205, etc. The computer has no way to know that it should skip from 145 to 200!

To see a way to fix this, suppose that we replace the line in Figure 9.11 that adds items to the menu with our generic "*Do a bit of interesting work using the counter.*" The resulting code is shown in Figure 9.14. Now, think about what sort of "interesting work" we need to perform if we want to produce a menu like that shown in Figure 9.13. Each time we perform this "interesting work" we should include four menu items for 0, 15, 30, and 45 minutes past the value in `hour`. This itself is a repetitive task. It can be performed by a counting loop that counts the number of minutes past the hour starting at 0 in steps of 15. A loop to do this might look like

```
int minutes = 0;
while ( minutes <= 45 ) {
    Do a bit of interesting work using the minute counter.
    minutes = minutes + 15;
}
```

Putting this fragment together with the template shown in Figure 9.14 gives us complete code to fill a menu similar to the Budget Rent-A-Car time menu. This code is shown in Figure 9.15.

```
JComboBox timeMenu = new JComboBox();

int hour = 100;
while ( hour <= 2400) {

    // Add four menu items in 15 minute increments
    int minutes = 0;
    while ( minutes <= 45 ) {
        timeMenu.addItem( "" + ( hour + minutes ) );
        minutes = minutes + 15;
    }

    hour = hour + 100;
}
```

Figure 9.15: Nested loops to create a 15-minute increment time menu

This code is an example of what is called *loop nesting*. The counting loop that increments `minutes` is referred to as the *inner loop* and the loop that increments `hours` is called the *outer loop*.

To reinforce the fact that we can use whatever statements are appropriate to do the "interesting work" in a `while` loop, let us consider one final variation of our time menu. For people who cannot figure out that 1800 hours is dinner time, we will write code to produce a menu with each time followed by the appropriate "AM" or "PM" suffix.

We will begin with a counting loop that simply counts hours in increments of one. That is, our basic template for this loop is

```
int hours = 1;
while ( hours <= 24 ) {
   Do a bit of interesting work using the hour counter.
   ++hours;
}
```

This time we have to do two bits of "interesting work" for each execution of the loop.

We have to use the value of `hour` to determine what number to display in the menu. When `hour` is 6, we should just display 6, but when `hour` is 16 we want to display 4. We can do this using a simple `if` statement. For example, we might say

```
int timeToDisplay = hour;
if ( timeToDisplay > 12 ) {
    timeToDisplay = timeToDisplay - 12;
}
```

We also have to decide whether to use the suffix "AM" or "PM". This is a bit trickier because midnight (i.e., 2400 hours) is considered "AM" while noon (i.e., 1200 hours) is considered "PM".

```
JComboBox timeMenu = new JComboBox();

int hour = 1;
while ( hour <= 24) {

    // Translate 24-hour time into 12 hour time
    int timeToDisplay = hour;
    if ( timeToDisplay > 12 ) {
        timeToDisplay = timeToDisplay - 12;
    }

    // determine the correct suffix and add to menu
    if ( 12 <= hour && hour < 24 ) {
        timeMenu.addItem( timeToDisplay + "PM" );
    } else {
        timeMenu.addItem( timeToDisplay + "AM" );
    }

    ++hour;
}
```

Figure 9.16: Nesting `if` statement in a loop to form a time menu

All the times from noon until just before midnight are considered "PM". Everything else is "AM". Therefore, the `if` statement

```
if ( 12 <= hour && hour < 24 ) {
    timeMenu.addItem( timeToDisplay + "PM" );
} else {
    timeMenu.addItem( timeToDisplay + "AM" );
}
```

will insert menu items with the appropriate suffixes. Putting these two `if` statements together with the loop template shown above gives us a complete loop to form the desired menu as shown in Figure 9.16.

9.5 Line by Line

Text documents, whether they be email messages, word processing documents, simple web pages, or the text of a Java program, have at least one level of structure that is quite repetitive. They are composed of a series of lines. This structure leads very naturally to loops that process the information in such documents by repeatedly performing one or more instructions to process each line. If the text to be processed is being received through a `NetConnection`, such loops will repeatedly invoke the `nextLine` method. Such loops are called *read loops* or *input loops*. We will consider the structure of read loops in this section. Later, you will see that read loops can also be used to process the lines of text stored in files on a machine's local disk.

To illustrate a very simple application of a read loop, we will revise one version of the SMTP client we explored in Chapter 4. Recall that when using the SMTP protocol, a client is expected to send a sequence of commands or requests starting with prefixes like "HELO", "MAIL FROM", "RCPT TO", etc. In response to each of these commands, the server sends a line indicating whether it was able to process the request successfully. These lines start with numeric codes like "250", which indicates success, and "501", which indicates an error in the request sent by the client.

The clients we constructed in Chapter 4 all displayed the responses received from the server in a `JTextArea` so that the user could verify that everything worked. We took several different approaches to accomplish this. In one version, we used the `nextLine` method to retrieve the server's response immediately after each client request. In another, we used the `dataAvailable` method to handle all server responses. In a third approach, we showed a version of the client that first sent all its requests to the server and then afterwards processed all of the server's responses together. We will use this third version as our starting point for the introduction of read loops. For your convenience, the `buttonClicked` method from this version is reproduced in Figure 9.17.

At this point, you know enough about loops that it should be obvious that we could use a `while` loop to replace the sequence of seven invocations of `log.append` and `connection.in.nextLine` that occur near the end of the method in Figure 9.17. A simple loop that counted from 1 to 7 and contained one copy of the line that invokes `nextLine` would do the trick. You certainly could write such a loop (and you probably should for practice). Java, however, provides an alternative to using a counting loop that is preferable in this situation.

There is a method named `hasNextLine` associated with the `in` stream of every `NetConnection` that can be used to determine how often to execute a read loop without counting lines. This method returns a `boolean` value. It returns `true` if a line has been received from the server and not yet retrieved by invoking `nextLine`. It returns `false` if the server has closed its side of the `NetConnection` and all the lines sent have been processed. Otherwise, it makes the program wait to see whether the server sends another line or closes the connection and then returns `true` or `false` as appropriate.

Using the `hasNextLine` method, we can replace the seven lines that invoke `nextLine` near the end of the code in Figure 9.17 with the shorter loop:

```
while ( connection.in.hasNextLine() ) {
    log.append( connection.in.nextLine() + "\n" );
}
```

Even though this loop is very short, it is worth thinking carefully about how it works. In our introduction to loops, we stressed that it is boring to do exactly the same thing over and over again. With counting loops, we avoided exact repetition by making the action performed by the loop body depend on a counter variable whose value changed each time the body was executed. The loop body in our short read loop does not contain any assignment statement. Its execution does not change the value of any variable. It does however, change the state of the `NetConnection` to the server. It is important to understand that each time `nextLine` is invoked, the system attempts to return a line that has not previously been accessed. Therefore, every time `nextLine` is invoked, it does something slightly different *by definition*.

Of course, read loops do not have to be as simple as this first example. Just as we can place many statements including nested loops and `if` statements within a counting loop, we can design read loops with more complex bodies if appropriate. For example, given that most of the messages sent by an SMTP server are fairly incomprehensible, we might decide to alter our program so that

```
// Send a message when the button is clicked
public void buttonClicked(  ) {
    // Establish a NetConnection and say hello to the server
    NetConnection connection = new NetConnection( SMTP_SERVER, SMTP_PORT );
    connection.out.println( "HELO " + this.getHostName() );

    // Send the TO and FROM addresses
    connection.out.println( "MAIL FROM: <" + from.getText() +">" );
    connection.out.println( "RCPT TO: <" + to.getText() +">" );

    // Send the message body
    connection.out.println( "DATA" );
    connection.out.println( message.getText() );
    connection.out.println( "." );

    // Terminate the SMTP session
    connection.out.println( "QUIT" );

    // Display the responses from the server
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );
    log.append( connection.in.nextLine() + "\n" );

    connection.close();
}
```

Figure 9.17: The `buttonClicked` method of a simple SMTP client

```
while ( connection.in.hasNextLine() ) {
    String response = connection.in.nextLine();
    if ( response.startsWith( "4" ) || response.startsWith( "5" ) ) {
        log.append( response + "\n" );
    }
}
```

Figure 9.18: A read loop for displaying SMTP error messages

it only shows the user lines that indicate an error has occurred. The codes that SMTP servers place at the beginning of each response are organized so that responses that indicate errors begin with a "4" or "5". Therefore, we can include an `if` statement in our read loop to selectively display only errors as shown in Figure 9.18.

In the original loop, we never associated a name with any of the lines received from the server. In the version shown in Figure 9.18, on the other hand, each line received is associated with the name `response` during the execution of the loop body that processes that line. This is essential. To emphasize this, consider the following code which will not work correctly because it invokes `nextLine` several times within the loop's body rather than associating a name with the result of a single invocation of `nextLine`.

```
// This is an example of INCORRECT CODE!!!
if ( connection.in.nextLine().startsWith( "4" ) ||
     connection.in.nextLine().startsWith( "5" ) ) {
    log.append( connection.in.nextLine() + "\n" );
}
```

Suppose that we replaced the body of the loop in Figure 9.18 with this `if` statement and then deliberately made a mistake when executing the program by entering "baduser@funnyplace.giv" as the sender address for our message. In this case, the sequence of responses sent by the server will look something like

```
220 smptserver.cs.williams.edu ESMTP Wed, 11 Jul 2007 11:07:04 -0400 (EDT)
250 smptserver.cs.williams.edu Hello 141.154.147.159, pleased to meet you
553 5.1.8 <baduser@funnyplace.giv> Domain of address baduser@funnyplace.giv does not exist
503 5.0.0 Need MAIL before RCPT
503 5.0.0 Need MAIL command
500 5.5.1 Command unrecognized: "This should not work at all"
500 5.5.1 Command unrecognized: "."
221 2.0.0 smptserver.cs.williams.edu closing connection
```

The correct version of the loop would display all but the first two and the last of these lines. Let us consider how the incorrect version would work.

When the loop body was first executed, it would retrieve the line

```
220 smptserver.cs.williams.edu ESMTP Wed, 11 Jul 2007 11:07:04 -0400 (EDT)
```

from the server and check to see if it started with "4". Since it does not start with "4" it would then evaluate

```
connection.in.nextLine().startsWith( "5" )
```

to see if it starts with "5". This condition, however, invokes `nextLine` again. Because each invocation of `nextLine` attempts to retrieve a new line from the `NetConnection`, rather than testing to see if the first line started with "5" it would actually check whether the second line received from the server:

```
250 smptserver.cs.williams.edu Hello 141.154.147.159, pleased to meet you
```

starts with "5". It does not, so the first execution of the loop body would terminate without displaying any line.

The second execution of the body would start by checking to see if the third line received

```
553 5.1.8 <baduser@funnyplace.giv> Domain of address baduser@funnyplace.giv does not exist
```

starts with "4". It does not, so the computer would continue checking the condition of the loop looking at the fourth line

```
503 5.0.0 Need MAIL before RCPT
```

to see if it starts with a "5". This condition will return `true`. Therefore the statement

```
log.append( connection.in.nextLine() + "\n" );
```

will be executed to display the line. Unfortunately, since this command also invokes `nextLine` it will not display the fourth line. Instead, it will access and display the fifth line

```
503 5.0.0 Need MAIL command
```

Luckily, this line is at least an error.

A very similar scenario will play out with even worse results on the next three lines. The statement in the loop body will test to see if the sixth line

```
500 5.5.1 Command unrecognized: "This should not work at all"
```

starts with a "4". Since it does not, it will continue to check if the seventh line

```
500 5.5.1 Command unrecognized: "."
```

starts with a "5". Since this test returns `true` it will then display the eight line

```
221 2.0.0 smptserver.cs.williams.edu closing connection
```

which is not even an error message from the server!

The basic lesson is that a read loop should almost always execute exactly one command that invokes `nextLine` during each execution of the loop body.

9.5.1 Sentinels

The `hasNextLine` method is not the only way to determine when a read loop should stop. In fact, in many situations it is not sufficient. The `hasNextLine` method does not return `false` until the server has closed its end of the connection. In many applications a client program needs to process a series of lines sent by the server and then continue to interact with the server by sending additional requests and receiving additional lines in response. Since the server cannot close the connection if it expects to process additional requests, protocols have to be designed to provide some other

way for a client to determine when to stop retrieving the lines sent in response to a single request. To illustrate how this is done, we will present components of a client based on one of the major protocols used to access email within the Internet, IMAP.

IMAP and SMTP share certain similar features. They are both text based protocols. In both protocols, most interactions consist of the client sending a line describing a request and the server sending one or more lines in response. IMAP, however, is more complex than SMTP. There are many more commands and options in IMAP than in SMTP. Luckily, we only need to consider a small subset of these commands in our examples:

LOGIN
: After connecting to a server, an IMAP client logs in by sending a command with the request code "LOGIN" followed by a user name and password.

SELECT
: IMAP organizes the messages it holds for a user into named folders or mailboxes. Before messages can be retrieved, the client must send a command to select one of these mailboxes. All users have a default mailbox named "inbox" used to hold newly arrived messages. A command of the form "SELECT inbox" can be used to select this mailbox.

FETCH
: Once a mailbox has been selected, the client can retrieve a message by sending a fetch request including the number of the desired message and a code indicating which components of the message are desired.

LOGOUT
: When a client wishes to disconnect from the server, it should first send a logout request.

Each request sent by an IMAP client must begin with a request identifier that is distinct from the identifiers used in all other requests sent by that client. Most clients form request identifiers by appending a sequence number to some prefix like "c" for "client". That is, client commands will typically start with a sequence of identifiers like "c1", "c2", "c3", etc.

Every response the server sends to the client also starts with a prefix. In many cases, this prefix is just a single "*". Such responses are called *untagged responses*. Other server responses are prefixed with the request identifier used as a prefix by the client in the request that triggered the response. Such responses are called *tagged responses*.

An example should make all of this fairly clear. Figure 9.19 shows the contents of an exchange in which a client connects to an IMAP server and fetches a single message from the user's inbox folder. To make the requests sent by the client stand out, we have drawn rectangles around each of them and displayed the text of these requests in bold-face italic font.[3]

The session begins with the client establishing a connection to port 143 on the IMAP server. As soon as such a connection is established, the server sends the untagged response "`* OK [CAPABILITY ...`" to the client.

Next, the client sends a LOGIN request and the server responds with a tagged request which, among other things, tells the client that the user identifier and password were accepted. The client request and the server's response to the login request are both tagged with "c1", a prefix chosen by the client.

[3]The contents of the session have also been edited very slightly to make things fit a bit better on the page and to hide the author's actual password.

```
* OK [CAPABILITY IMAP4REV1 STARTTLS AUTH=LOGIN] 2004.357 at Wed, 11 Jul 2007 15:53:00 -0400 (EDT)
c1 LOGIN tom somethingSecret
c1 OK [CAPABILITY IMAP4REV1 NAMESPACE UNSELECT SCAN SORT MULTIAPPEND] User tom authenticated
c2 SELECT inbox
* 549 EXISTS
* 0 RECENT
* OK [UIDVALIDITY 1184181410] UID validity status
* OK [UIDNEXT 112772] Predicted next UID
* FLAGS (Forwarded Junk NotJunk Redirected $Forwarded \Answered \Flagged \Deleted \Draft \Seen)
c2 OK [READ-WRITE] SELECT completed
c3 FETCH 81 (BODY[])
* 81 FETCH (BODY[] {532}
Return-Path: <sales@lacie.com>
Received: from mail.inherent.com (lacie1.inherent.com [207.173.32.81])
        by ivanova.inherent.com (8.10.1/8.10.1) with ESMTP id g0OGhZh12957
        for <tom@cs.williams.edu>; Thu, 24 Jan 2007 08:43:35 -0800
Message-Id: <200201241643.g0OGhZh12957@ivanova.inherent.com>
Content-type: text/plain
Date: Thu, 24 Jan 2007 08:42:46 -0800
From: sales@lacie.com
Subject: Thanks for shopping LaCie
To: tom@cs.williams.edu

Your order has been received and will be processed.
Thanks for shopping LaCie.

)
c3 OK FETCH completed
c4 LOGOUT
* BYE bull IMAP4rev1 server terminating connection
c4 OK LOGOUT completed
```

Figure 9.19: A short IMAP client/server interaction

Figure 9.20: Interface for an ugly (but simple) IMAP client

The client then sends a SELECT request to select the standard inbox folder. The server has a bit more to say about this request. It sends back a series of untagged responses providing information including the number of messages in the folder, 549, and the number of new messages, 0. Finally, after five such untagged responses, it sends a response tagged with the prefix "c2" to tell the client that the SELECT request has been completed successfully.

The client's third request is a FETCH for message number 81. The server responds to this request with two responses. The first is an untagged response that is interesting because it spans multiple lines. It starts with "* 81 FETCH (BODY[] {532}", includes the entire contents of the requested email message and ends with a line containing a single closing parenthesis. It is followed by a tagged response that indicates that the fetch was completed.

Finally, the client sends a LOGOUT request. Again the server responds with both an untagged and a tagged response.

The aspects of this protocol that are interesting here are that the lengths of the server responses vary considerably and that the server does not indicate the end of a response by closing the connection. You may have noticed, however, that the server does end each sequence of responses with a response that is fairly easy to identify. The last line sent in response to each of the client requests is a tagged response starting with the identifier the client used as a prefix in its request. Such a distinguished element that indicates the end of a sequence of related input is called a *sentinel*.

To illustrate how to use a sentinel, suppose we want to write an IMAP client with a very

primitive interface. This client will simply provide the user with the ability to enter account information and a message number. After this is done, the user can press a "Get Message" button and the program will attempt to fetch the requested message by sending requests similar to those seen in Figure 9.19. As it does this, the client will display all of the lines exchanged with the server. A nicer client would only display the message requested, but this interface will enable us to keep our loops a bit simpler. An image of how this interface might look is shown in Figure 9.20.

We will assume that the program includes declarations for the following four variables:

int requestCount;
To keep track of the number of requests that have been sent to the server.

String requestID;
To refer to the request identifier placed as a prefix on the last request sent to the server.

JTextArea display;
To refer to the `JTextArea` in which the dialog with the server is displayed.

NetConnection toServer;
To refer to the connection with the server.

Let us consider the client code required to send the "SELECT" request to the server and display the responses received. Sending this command is quite simple since it does not depend on the account information or message number the user entered. Processing the responses, on the other hand, is interesting because the number of responses the server sends may vary. For example, if you compare the text in Figure 9.19 to that displayed in the window shown in Figure 9.20 you will notice that if a new message has arrived, the server may add a response to tell the client about it.

The code to send the select request might look like this:

```
++requestCount;
requestID = "c" + requestCount;

toServer.out.println( requestID + " SELECT inbox" );
display.append( requestID + " SELECT inbox" + "\n" );
```

The first two lines determine the request identifier that will be included as a prefix. The next line sends the request through the `NetConnection`. The final line adds it to the text area.

After sending this request, the client should retrieve all the lines sent by the server until the first line tagged with the prefix the client included in the select request. This prefix is associated with the variable `requestID`. Since the number of requests can vary, we should clearly use a read loop. One might expect the loop to look something like

```
// Warning:
//
//         This loop will not work!!
//
while ( ! responseLine.startsWith( requestID ) ) {
    responseLine = toServer.in.nextLine();
    display.append( responseLine  + "\n" );
}
```

Unfortunately, this will not work. The variable `responseLine` is assigned its first value when the loop's body is first executed. The condition in the loop header, however, will be executed before every execution of the loop body, including the first. That means the condition will be evaluated before any value has been associated with `responseLine`. Since the condition involves `responseLine`, this will lead to a program error.

You might have heard the expression "prime the pump." It refers to the process of pouring a little liquid into a pump before using it to replace any air in the pipes with water so that the pump can function. Similarly, to enable our loop to test for the sentinel the first time, we must "prime" the loop by reading the first line of input before the loop. This means that the first time the loop body is executed, the line of input it should process (i.e., add to `display`) will already be associated with `responseLine`. To make the loop work, we need to design the loop body so that this is true for all executions of the loop. We do this by writing a loop body that first processes one line and then retrieves the next line so that it can be processed by the next execution of the loop body. The resulting loop looks like

```
String responseLine = toServer.in.nextLine()";
while ( ! responseLine.startsWith( requestID ) ) {
    display.append( responseLine  + "\n" );
    responseLine = toServer.in.nextLine();
}
```

The original loop performed two basic steps: retrieving a line from the server and displaying the line on the screen. We prime the loop by performing one of these steps before its first execution. As a result, we also need to finish the loop's work by performing the other step once after the loop completes. To appreciate this, consider what will happen to the last line processed. This line will start with the sentinel value. It will be retrieved by the last execution of the second line in the loop body

```
responseLine = toServer.in.nextLine();
```

The computer will then test the condition in the loop header and conclude that the loop should not be executed again. As a result, the instruction inside the loop that appends lines to the `display` will never be executed for the sentinel line. If we want the sentinel line displayed, we must add a separate command to do this after the loop. Following this observation, complete code for sending the select request and displaying the response received is shown in Figure 9.21.

9.6 Accumulating Loops

The code shown in Figure 9.21 includes everything needed to process a select request. To complete our IMAP client, however, we also need code to handle the login, fetch and logout requests. The code to handle these requests would be quite similar to that for the select request. We should exploit these similarities. Rather than simply writing code to handle each type of request separately, we should write a `private` helper method to perform the common steps.

In particular, we could write a method named `getServerResponses` that would retrieve all of the responses sent by the server up to and including the tagged response indicating the completion of the client's request, and return all of these responses together as a single `String`. This method would take the `NetConnection` and the request identifier as parameters. Given such a method, we could rewrite the code for a select request as

```
++requestCount;
requestID = "c" + requestCount;

toServer.out.println( requestID + " SELECT inbox" );
display.append( requestID + " SELECT inbox" + "\n" );

display.append( getServerResponses( toServer, requestID ) );
```

Preliminary code for a `getServerResponses` method is shown in Figure 9.22. The method includes a loop very similar to the one we included in Figure 9.21. This loop, however, does not put the lines it retrieves into a `JTextArea`. Instead, it combines them to form a single `String`. A variable named `allResponses` refers to this `String`.

The variable `allResponses` behaves somewhat like the counters we have seen in earlier loops. It is associated with the empty string as an initial value before the loop just as an `int` counter variable might be initialized to zero. Then, in the loop body we "add" to its contents by concatenating the latest line received to `allResponses`. We are, however, doing something more than counting. We are accumulating the information that will serve as the result of executing the loop. As a result, such variables are called *accumulators*.

Of course, we can accumulate things other than `String`s. To illustrate this, we will use an accumulator to fix a weakness in the approach we have taken to implementing our IMAP client.

Suppose that we use our IMAP client to retrieve an email message about the Star Wars movies including the following text (adapted from the IMDB web site):

```
A New Hope opens with a rebel ship being boarded by the tyrannical
Darth Vader. The plot then follows the life of a simple farmboy, Luke
Skywalker, as he and his newly met allies (Han Solo, Chewbacca, Ben Kenobi,
c3-po, r2-d2) attempt to rescue a rebel leader, Princess Leia, from the
clutches of the Empire. The conclusion is culminated as the Rebels,
including Skywalker and flying ace Wedge Antilles make an attack on the
Empires most powerful and ominous weapon, the Death Star.
```

Can you see why retrieving this particular message might cause a problem?

The IMAP client we have written uses the sequence "c1", "c2", "c3", etc. as its request identifiers. The request identifier used in the request to fetch the message will always be "c3". The fourth line of the text shown above starts with the characters "c3" as part of the robot name "c3-po". As a result, the loop in `getServerResponses` will terminate when it sees this line, failing to retrieve the rest of the message and several other lines sent by the server. An error like this might seem unlikely to occur in practice, but the designers of IMAP were concerned enough to include a mechanism to avert the problem in the rules of the protocol.

If you examine the first line of the untagged response to the fetch request shown in Figure 9.19:

```
* 81 FETCH (BODY[] {532}
```

```
    // Determine the request identifier
    ++requestCount;
    requestID = "c" + requestCount;

    // Send and display the request
    toServer.out.println( requestID + " SELECT inbox" );
    display.append( requestID + " SELECT inbox" + "\n" );

    // Prime the loop by retrieving the first response
    String responseLine = toServer.in.nextLine();

    // Retrieve responses until the sentinel is received
    while ( ! responseLine.startsWith( requestID ) ) {
        display.append( responseLine  + "\n" );
        responseLine = toServer.in.nextLine();
    }

    // Display the final response
    display.append( responseLine + "\n" );
```

Figure 9.21: Client code to process an IMAP "SELECT inbox" request

```
// Retrieve all responses the server sends for a single request
public String getServerResponses( NetConnection toServer, String requestID ) {
    String allResponses = "";

    String responseLine = toServer.in.nextLine();
    while ( ! responseLine.startsWith( requestID ) ) {
        allResponses = allResponses + responseLine  + "\n";
        responseLine = toServer.in.nextLine();
    }
    allResponses = allResponses + responseLine  + "\n";

    return allResponses;
}
```

Figure 9.22: A method to collect an IMAP server's responses to a single client request

you will notice that it ends with the number 532 in curly braces. This number is the total length of the text of the email message that follows. According to the rules of IMAP, when a server wants to send a response that spans multiple lines, the first line must end with a count of the total size of the following lines. When the client receives such a response, it retrieves lines without checking for the sentinel value until the total number of characters retrieved equals the number found between the curly braces. In the IMAP protocol description, such collections of text are called literals. The only trick is that the count includes not just the characters you can see, but additional symbols that are sent through the network to indicate where line breaks occur. There are two such invisible symbols for each line break known as the "return" and "new line" symbols. Both must be included at the end of each line sent through the network.

Figure 9.23 shows a revised version of the `getServerResponses` method designed to handle IMAP literals correctly. At first, it looks very different from the preceding version of the method, but its basic structure is the same. The body of the `while` loop in the original version of the method contained just two instructions:

```
allResponses = allResponses + responseLine + "\n";
responseLine = toServer.in.nextLine();
```

The new version retains these two instructions as the first and last instructions in the main loop. However, it also inserts one extra instruction — a large `if` statement designed to handle literals — between them. The condition of the `if` statement checks for literals by seeing if the first line of a server response ends with a "}". If no literals are sent by the server, the body of this `if` statement is skipped and the loop works just like the earlier version.

If the condition in the `if` statement is `true` then the body of the `if` statement is executed to process the literal. The `if` statement contains a nested `while` loop. With each iteration of this loop, a new line sent by the server is processed. Two types of information about each line are accumulated. The line

```
allResponses = allResponses + responseLine + "\n";
```

which is identical to the first line of the outer loops, continues the process of accumulating the entire text of the server's response in the variable `allResponses`. The line

```
charsCollected = charsCollected + responseLine.length() + LINE_END;
```

accumulates the total length of all of the lines of the literal that have been processed so far in the variable `charsCollected`. The value `NEW_LINE` accounts for the two invisible characters transmitted to indicate the end of each line sent. `charsCollected` is initialized to 0 before the loop to reflect the fact that initially no characters in the literal have been processed.

The instructions that precede the inner loop extract the server's count of the number of characters it expects to send from the curly braces at the end of the first line of the response. The header of the inner loop uses this information to determine when the loop should stop. At each iteration, this condition compares the value of `charsCollected` to the server's prediction and stops when they become equal.

9.7 String Processing Loops

`Strings` frequently have repetitive structures. Inherently, every string is a sequence of characters. Some strings can be interpreted as sequences of words or larger units like sentences. When we need

```
// Retrieve all responses the server sends for a single request
public String getServerResponses( NetConnection toServer, String requestID ) {
   // Number of invisible symbols present between each pair of lines
   final int LINE_END  = 2;

   String allResponses = "";
   String responseLine = toServer.in.nextLine();

   while ( ! responseLine.startsWith( requestID ) ) {

      allResponses = allResponses + responseLine  + "\n";

      // Check for responses containing literal text
      if ( responseLine.endsWith( "}" ) ) {

         // Extract predicted length of literal text from first line of response
         int lengthStart = responseLine.lastIndexOf( "{" ) + 1;
         int lengthEnd = responseLine.length() - 1;
         String length = responseLine.substring( lengthStart, lengthEnd );
         int promisedLength = Integer.parseInt( length );

         // Used to count characters of literal as they are retrieved
         int charsCollected = 0;

         // Add lines to response until their length equals server's prediction
         while ( charsCollected < promisedLength ) {
            responseLine = toServer.in.nextLine();
            allResponses = allResponses + responseLine + "\n";
            charsCollected = charsCollected + responseLine.length() + LINE_END;
         }

     }

      // Get the next line following a single line response or a literal
      responseLine = toServer.in.nextLine();
   }

   return allResponses + responseLine + "\n";
}
```

Figure 9.23: A method to correctly retrieve responses including literals

to process a `String` in a way that involves such repetitive structure, it is common to use loops. In this section, we introduce some basic techniques for writing such loops.

Our first example involves a public confession. For years, I pestered my youngest daughter about a punctuation "mistake" she made when typing. The issue involved the spacing after periods. In the olden days, when college-bound students left home with a typewriter rather than a laptop, they (we?) were taught to place two spaces after the period at the end of a sentence. In the world of typewriters in which all characters had the same width, this rule apparently improved readability. Once word processors and computers capable of using variable-width fonts arrived, this rule became unnecessary and students were told to place just one space after each sentence. Unfortunately, no one told me that the rule had changed. I was trying to teach my daughter the wrong rule!

To make up for my mistake, let's think about how we could write Java code to take a `String` typed the old way and bring it into the present by replacing all the double spaces after periods with single spaces. In particular, we will define a method named `reduceSpaces` that takes a `String` in which some or all periods may be followed by double spaces and returns a `String` in which any sequence of more than one space after a period has been replaced by a single space. Even if you have always typed the correct number of spaces after your periods, this example illustrates techniques that can be used to revise the contents of a `String` in many other useful ways.

A good way to start writing a loop that repeats some process is to write the code to perform the process once. Generally speaking, if you do not know how to do something just once, you are not going to be able to do it over and over again. In our case, "the process" is finding a period followed by two spaces somewhere in a `String` and replacing the two spaces with a single space.

As our English description suggests, the first step is to find a period followed by two spaces. This can be done using the `indexOf` method. Assuming that the `String` to be processed is associated with a variable named `text`, then the statement

```
int periodPosition = text.indexOf( ".  " );
```

associates name `periodPosition` with the position of the first period within `text` that is followed by two spaces. Since it is probably a bit difficult to count the spaces between the quotes in this code, in the remainder of our presentation, we will use the equivalent expression

```
"." + " " + " "
```

in place of `".  "`.

Once `indexOf` tells us where the first space that should be eliminated is located, we can use the `substring` method to break the `String` into two parts: the characters that appear before the extra space and the characters that appear after that space as follows

```
int periodPosition = text.indexOf( "." + " " + " " );
String before = text.substring( 0, periodPosition + 2 );
String after = text.substring( periodPosition + 3 );
```

Then, we can associate the name `text` with the `String` obtained by removing the second space by executing the assignment

```
text = before + after;
```

With this code to remove a single extra space, we can easily construct a loop and a method to remove all of the extra spaces. A version of `reduceSpaces` based on this code is shown in

```
private String reduceSpaces( String text ) {

    while ( text.contains( "." + " " + " " ) ) {
        int periodPosition = text.indexOf( "." + " " + " " );
        String before = text.substring( 0, periodPosition + 2 );
        String after = text.substring( periodPosition + 3 );
        text = before +  after;
    }
    return text;
}
```

Figure 9.24: A method to compress double spaces after periods

Figure 9.24. We have simply placed the instructions to remove one double space in a loop whose condition states that those instructions should be executed repeatedly as long as `text` still contains at least one double space after a period.

Suppose instead that we were given text in which each sentence was ended with a period followed by a single space and we wanted to convert this text into "typewriter" format where every period was followed by two spaces. Again, we start by writing instructions to add an extra space after just one of the periods in the text. The following instructions do the job by splitting the text into "before" and "after" components right after the first period:

```
int periodPosition = text.indexOf( "." + " " );
String before = text.substring( 0, periodPosition + 1 );
String after = text.substring( periodPosition + 1 );
text = before + " " + after;
```

However, if we put these instruction inside a loop with the header

```
while ( text.contains( "." + " " ) ) {
```

the resulting code won't add extra spaces after all of the periods. Instead, it will repeatedly add extra spaces after the first period over and over again forever.

To understand why this loop won't work, note that even after we replace a copy of the substring `"." + " "` in `text` with the longer sequence `"." + " " + " "` , `text` still contains a copy of `"." + " "` at exactly the spot where the replacement was made. This interferes with the correct operation of the proposed code in two ways. First, if the condition in the loop header is `true` before we execute the body of the loop, it will still be `true` afterwards. This means that the loop will never stop. If you run such a program, your computer will appear to lock up. This is an example of what is called an *infinite loop*. Luckily, most program development environments provide a convenient way to terminate a program that is stuck in such a loop.

In addition, each time the loop body is executed the invocation of `indexOf` in the first line will find the same period. The first time, there will only be one space after this period. The second time there will be two spaces, then three and so on. Not only does this loop execute without stopping, as it executes, it only changes the number of spaces after the first period.

There are two ways to solve these problems. The first is to take advantage of the fact that `indexOf` accepts a second, optional parameter telling it where to start its search. If we always start

```
int periodPos = -1;
while ( text.indexOf( "." + " ", periodPos + 1 ) != -1  ) {
    periodPos = text.indexOf( "." + " ", periodPos + 1 );
    String before = text.substring( 0, periodPos + 1 );
    String after = text.substring( periodPos + 1  );
    text = before + " " + after;
}
```

Figure 9.25: An inefficient loop to add spaces after periods

the search after the last period we processed, each period will only be processed once. Also, recall that `indexOf` returns `-1` if it cannot find a copy of the search string. Therefore, if we replace the use of the `contains` method with a test to see if an invocation of `indexOf` returned `-1`, our loop will stop correctly after extra blanks are inserted after all of the periods. A correct (but inefficient) way to do this is shown in Figure 9.25.

The body of this loop is identical to the preceding version, except that `periodPos + 1` is included as a second parameter to the invocation of `indexOf`. This ensures that each time the computer looks for a period it starts after the position of the period processed the last time the loop body was executed. To make this work correctly for the first iteration, we initially associate -1 with the `periodPos` variable.

The other difference between this loop and the previous one is the condition. Instead of depending on `contains`, which always searches from the beginning of a `String`, we use an invocation of `indexOf` that only searches after the last period processed. This version is correct but inefficient. Having replaced `contains` with `indexOf`, the loop now invokes `indexOf` twice in a row with exactly the same parameters producing exactly the same result each time the body is executed.

Efficiency is a tricky subject. Computers are very fast. In many cases, it really does not matter if the code we write does not accomplish its purpose in the fewest possible steps. The computer will do the steps so quickly no one will notice the difference anyway. When working with loops, however, a few extra steps in each iteration can turn into thousands or millions of extra steps if the loop body is repeated frequently. As a result, it is worth taking a little time to avoid using `indexOf` to ask the computer to do exactly the same search twice each time this loop executes.

The way to make this loop more efficient is to "prime" the loop much as we primed read loops. In this case, we prime the loop by performing the first `indexOf` before the loop and saving the result in a variable. In addition, we will need to perform the `indexOf` needed to set this variable's value as the last step of each execution of the loop body in preparation for the next evaluation of the loop's condition. We can then safely test this variable in the loop condition. An `addSpaces` method that uses this approach is shown in Figure 9.26.

If you think about it for a moment, you should realize that the same inefficiency we identified in Figure 9.25 exists in the code shown for the `reduceSpaces` method in Figure 9.24. It is not as obvious in the `reduceSpaces` method because we do not invoke `indexOf` twice in a row. The invocations of `contains` and `indexOf` in this method, however, make the computer search through the `String` twice in a row looking for the same substring. Worse yet, both searches start from the very beginning of the text, rather than after the last period processed. As a result, it would be more efficient to rewrite `reduceSpaces` in the style of Figure 9.26. We encourage the reader to sketch out such a revision of the method.

```
private String addSpaces( String text ) {

    int periodPos = text.indexOf( "." + " " );
    while ( periodPos >= 0 ) {
        String before = text.substring( 0, periodPos + 1 );
        String after = text.substring( periodPos + 1  );
        text = before + " " + after;
        periodPos = text.indexOf( "." + " ", periodPos + 1 );
    }
    return text;
}
```

Figure 9.26: A method to place double spaces after periods

An alternative approach to the problem of implementing `addSpaces` is to use a variable that accumulates the result as we process periods. Suppose we call this variable `processed` and declare it as

```
String processed = "";
```

The loop body will now function by moving segments of the original `String` from the variable `text` to `processed` fixing the spacing after one period in each segment copied.

Again, let's start by thinking about how to do this just once. In fact, to make things even simpler, just think about doing it for the first period in the `String`. The code might look like

```
int periodPos = text.indexOf( "." + " " );
processed = text.substring( 0, periodPos + 2 ) + " ";
text = text.substring( periodPos + 2  );
```

We find the period, and then we move everything up to and including the period and following spaces to `processed` while leaving everything after the space following the first period in `text`.

Now, all we need to do to make this work for periods in the middle of the text, rather than just the first period, is to add the prefix from `text` to `processed` rather than simply assigning it to `processed`. That is, we change the assignment

```
processed = text.substring( 0, periodPos + 2 ) + " ";
```

to be

```
processed = processed + text.substring( 0, periodPos + 2 ) + " ";
```

We still want to avoid searching the text in both the loop condition and the loop body. Therefore, we prime the loop by searching for the first period and repeat the search as the last step of the loop body. The complete loop is shown in Figure 9.27. Note that any suffix of the original text after the last period will remain associated with the variable `text` after the loop terminates. Therefore, we have to concatenate `processed` with `text` to determine the correct value to return.

The behavior of the values associated with `text` by this loop is in some sense the opposite of that of the values of `processed`. In the case of `processed`, information is accumulated by

```
private String addSpaces( String text ) {
    String processed = "";

    int periodPos = text.indexOf( "." + " " );
    while ( periodPos >= 0 ) {
        processed = processed + text.substring( 0, periodPos + 2 ) + " ";
        text = text.substring( periodPos + 2  );
        periodPos = text.indexOf( "." + " " );
    }
    return processed + text;
}
```

Figure 9.27: Another method to place double spaces after periods

gathering characters together. As the loop progresses, on the other hand, the contents of `text` are gradually dissipated by throwing away characters. In fact, it is frequently useful to design loops that gradually discard parts of a `String` as those parts are processed and terminate when nothing of the original `String` remains.

As an example of this, consider how to add one useful feature to the SMTP client that was introduced in Chapter 4 and discussed again in Section 9.5. That program allows a user to send an email address to a single destination at a time. Most real email clients allow one to enter an entire list of destination addresses when sending a message. These clients send a copy of the message to each destination specified. Such clients allow the user to type in all of the destination addresses on one line separated from one another by spaces or commas. An example of such an interface is shown in Figure 9.28

The SMTP protocol supports multiple destinations for a message, but it expects each destination to be specified as a separate line rather than accepting a list of multiple destinations in one line. We have seen that an SMTP client typically transmits a single message by sending the sequence of commands "HELO", "MAIL FROM", "RCPT TO", "DATA", and "QUIT" to the server. To specify multiple destinations the client has to send multiple "RCPT TO" commands so that there is one such command for each destinations address.

Our goal is to write a loop that will take a `String` containing a list of destinations and send the appropriate sequence of "RCPT TO" commands to the server. To keep our code simple, we assume that exactly one space appears between each pair of addresses.

Once again, start by thinking about the code we would use to send just one of the "RCPT TO" commands to the server. Assuming that the variable `destinations` holds the complete `String` of email addresses and that `connection` is the name of the `NetConnection` to the server, we could use the instructions

```
int endOfAddress = destinations.indexOf( " " );
String destination = destinations.substring( 0, endOfAddress);
connection.out.println( "RCPT TO: <" + destination +">" );
```

The first of these instructions finds the space after the first email address. The second uses the position of this space with the `substring` method to extract the first address from `destinations`. The final line sends the appropriate command to the server.

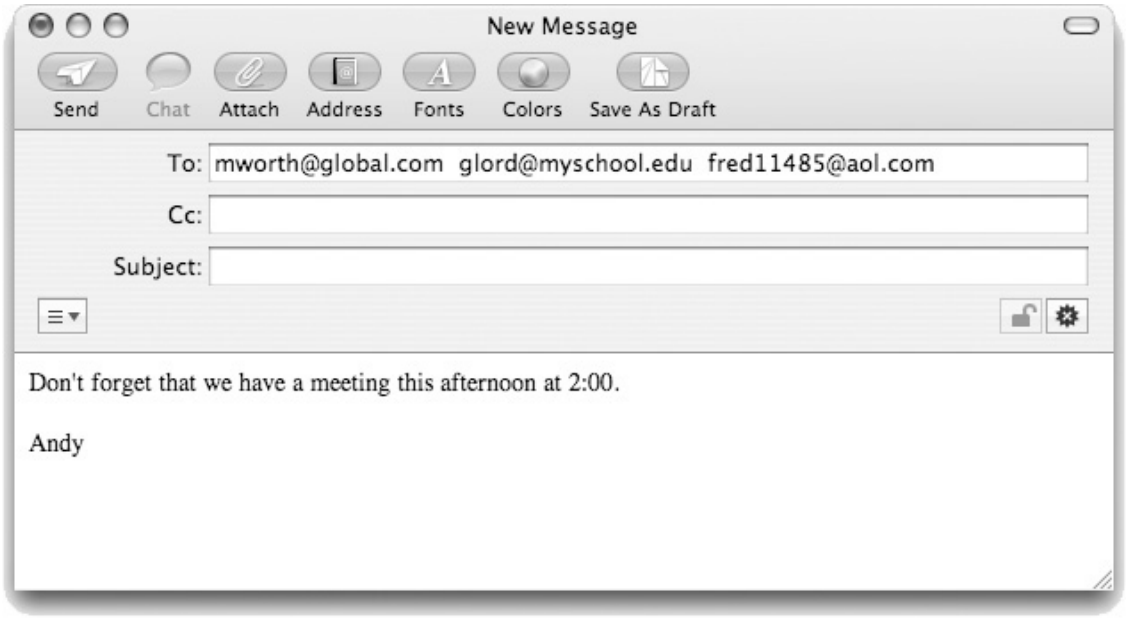

Figure 9.28: Typical interface for specifying multiple email destinations

We clearly cannot simply repeat these commands over and over to handle multiple destinations. If we repeat just these instructions, the first line will always find the same space and the loop will just send identical copies of the same "RCPT TO" command forever:

```
RCPT TO: <mworth@global.com>
RCPT TO: <mworth@global.com>
RCPT TO: <mworth@global.com>
   . . .
```

We can get much closer to what we want by adding the following line to the end of the code we have already written

```
destinations = destinations.substring( endOfAddress + 1 );
```

This line removes the address processed by the first three lines from the value of `destinations`. Therefore, if we execute these four lines repeatedly, the program sends a "RCPT TO" for the first address in `destinations` the first time, a command for the second address during the second execution, and so on:

```
RCPT TO: <mworth@global.com>
RCPT TO: <glord@myschool.edu>
   . . .
```

The last address, however, will be a problem. When all but one address has been removed from `destinations`, there will be no spaces left in the `String`. As a result, `indexOf` will return -1 as the value to be associated with `endOfAddress`. Invoking substring with -1 as a parameter will cause a program error.

```
// Warning: This code is correct, but a bit inelegant
while ( destinations.length() > 0 ) {
    int endOfAddress = destinations.indexOf( " " );
    if ( endOfAddress != -1 ) {
        String destination = destinations.substring( 0, endOfAddress);
        connection.out.println( "RCPT TO: <" + destination +">" );
        destinations = destinations.substring( endOfAddress + 1 );
    } else {
        connection.out.println( "RCPT TO: <" + destinations +">" );
        destinations = "";
    }
}
```

Figure 9.29: An awkward loop to process multiple destinations

One way to address this problem is to place the statements that depend upon the value that `indexOf` returns within a new `if` statement that first checks to see if the value returned was -1. A sample of how this might be done is shown in Figure 9.29. The code in the first branch of this `if` statement handles all but the last address in a list by extracting one address from the list using the value returned by `indexOf`. The code in the second branch is only used for the last address in the list. It simply treats the entire contents of `destinations` as a single address and then replaces this one element list with the empty string.

While the code shown in Figure 9.29 will work, it should bother you a bit. The whole point of a loop is to tell the computer to execute certain instructions repeatedly. Given the loop in Figure 9.29, we know that the statements in the `else` part of the `if` statement will not be executed repeatedly. The code is designed to ensure that these statements will only be executed once each time control reaches the loop. What are instructions that are not supposed to be repeated doing in a loop? This oddity is a clue that the code shown in Figure 9.29 probably is not the most elegant way to solve this problem.

There is an alternate way to solve this problem that is so simple it will seem like cheating. Accordingly, we want to give you a deep and meaningful explanation of the underlying idea before we show you its simple details.

When dealing with text that can be thought of as a list, it is common to use some symbol to indicate where one list element ends and the next begins. This is true outside of the world of Java programs. In English, there are several punctuation marks used to separate list items. These include commas, semi-colons, and periods. The preceding sentence show a nice example of how commas are used in such lists. Also, this entire paragraph can be seen as a list of sentences where each element of this list is separated from the next by a period.

There is, however, an important difference between commas and periods in English. The comma is a *separator*. We place separators between the elements of a list. The period is a terminator. We place one after each element in a list, including the last element. When a separator is used to delimit list items, there is always one more list item than there are separators. When a terminator is used the number of list items equals the number of terminators.

This simple difference can lead to awkwardness when we try to write Java code to process text that contains a list in which separators are used as delimiters. The loop is likely to work by

```
destinations = destinations + " ";
while ( destinations.length() > 0 ) {
    int endOfAddress = destinations.indexOf( " " );
    String destination = destinations.substring( 0, endOfAddress);

    connection.out.println( "RCPT TO: <" + destination +">" );

    destinations = destinations.substring( endOfAddress + 1 );
}
```

Figure 9.30: A simpler loop to process multiple email destination addresses

searching for the delimiter symbols. If a separator is being used, there will still be one item left to process after the last separator has been found and removed. This is the difficulty the code in Figure 9.29 was designed to handle. The spaces that separate the email addresses are used as separators.

We can fix this by using spaces to terminate the email addresses rather than separate them. It would be odd to make someone using our program type in an extra space after the last address, but it is quite easy to write code to add such an extra space. Once such a space is added, we can treat the spaces as terminators instead of separators. This make it unnecessary to treat the last address as a special case. Figure 9.30 shows the complete code to handle a list of destination addresses using this approach.

9.8 Summary

This chapter is different from the chapters that have preceded it in an important way. In each of the earlier chapters, we introduced several new features of the Java language. In this chapter, we only introduced one. Strangely, this chapter is not any shorter than the others. This reflects the fact that while the technical details of the syntax and interpretation of a `while` loop in Java are rather simple, learning to write loops effectively is not. Therefore, although we introduced all of the rules of Java that govern `while` loops in Section 9.2, we filled several other sections with examples designed to introduce you to important techniques and common patterns for constructing loops.

As we tried to stress in several of our examples, it is frequently helpful to write complete code to perform a task once before seriously thinking about how a loop can be used to do it repeatedly. Having concrete code at hand to perform an operation once can expose details you have to understand in order to incorporate the code into a loop.

Technically, a loop only has two parts, its header and its body. In many of the loops we discussed, however, it was possible to identify four parts. The first part is the statement or statements preceding the loop that are designed to initialize the variables that are critical to the loops operation or prepare other aspects of the computer's state for the loop. For counting loops, this simply involved initializing a numeric variable. For read loops, we saw that we sometimes prime a loop by retrieving the first input from its source before the loop body.

The loop condition is the second component of all loops. It is crucial to ensure that repeatedly executing the body of the loop eventually leads to a point where this condition is `false`. Otherwise,

the result will be an infinite loop. It is also important to realize that the condition is tested just before every execution of the loop body including the first. This is critical when deciding how to initialize variables before the loop.

The loop body can often be divided into two subparts. One subset of the instructions in the loop body can be identified as commands designed to do the actual work of the loop. The remaining instructions are designed to alter program variables so that the next execution of the loop will correctly move on to the next step required. Thus, almost all while loops can be abstracted to fit the following template

```
// Initialize loop variables and other state
while ( // loop variable values indicate more work is left ) {
    // Do some interesting work
    // Update the loop variables
}
```

This template can also serve as a very useful guide when constructing loops.

Important patterns are also seen in the ways loop variables are manipulated. We have seen examples of several of these including counter variables, accumulators, and `String` variables whose values are gradually reduced to the empty string during loop processing.