Appendix 2A

Text 1 – Coding of Sentences

4.1	INTRODUCTION
T1/INT/S1	Suppose that you need to *print* a *string* (e.g., "Welcome to *Java*!") a hundred times.
T1/INT/S2	It would be tedious to have to write the following *statement* a hundred times:
T1/INT/S3	*Java* provides a powerful control structure called a *loop* that controls how many times an operation or a sequence of operations is performed in succession.
T1/INT/S4	Using a *loop statement*, you simply tell the computer to *print* a *string* a hundred times without having to code the *print statement* a hundred times.
T1/INT/S5	*Loops* are structures that control repeated executions of a block of *statements*.
T1/INT/S6	The concept of *looping* is fundamental to programming.
T1/INT/S7	*Java* provides three types of *loop statements*: *while loops*, *do-while loops*, and *for loops*.
4.2	THE *WHILE LOOP*
T1/WL/S1	The syntax *for* the *while loop* is as follows:
T1/WL/S2	The *while loop* flow chart is shown in *Figure 4.1(a).*
T1/WL/S3	The part of the *loop* that contains the *statements* to be repeated is called the *loop body*.
T1/WL/S4	A one-time execution of a *loop* body is referred to as an iteration of the *loop*.
T1/WL/S5	Each *loop* contains a *loop-continuation* condition, a Boolean expression that controls the execution of the body.
T1/WL/S6	It is always evaluated be*fore* the *loop* body is executed.
T1/WL/S7	If its evaluation is true, the *loop* body is executed.
T1/WL/S8	If its evaluation is false, the entire *loop* terminates and the program control turns to the *statement* that follows the *while loop*.
T1/WL/S9	For example, the following *while loop* prints "Welcome to *Java*!" a hundred times.
T1/WL/S10	The flow chart of the preceding *statement* is shown in *Figure 4.1(b).*
T1/WL/S11	The variable *count* is initially 0.
T1/WL/S12	The *loop* checks whether (*count* < *100*) is true.
T1/WL/S13	If so, it executes the *loop* body to *print* the message "Welcome to *Java*!" and increments *count* by 1.
T1/WL/S14	It repeatedly executes the *loop* body until (*count* < *100*) becomes false.
T1/WL/S15	When (*count* < *100*) is false (i.e., when *count* reaches *100*), the *loop* terminates and the next *statement* after the *loop statement* is executed.
4.2.1	AN ADVANCED MATH LEARNING TOOL
T1/AMLT/S1	The Math subtraction learning tool program in *Listing* 3.5, *SubtractionTutor.Java*, generates just one question for each run.
T1/AMLT/S2	You can use a *loop* to generate questions repeatedly.
T1/AMLT/S3	*Listing* 4.1 gives a program that generates ten questions and reports the number of correct answers after a student answers all ten questions.
T1/AMLT/S4	The program also displays the time spent on the test and lists all the questions, as shown in *Figure 4.2*.

T1/AMLT/S5	The program uses the control variable *count* to control the execution of the *loop*.
T1/AMLT/S6	*Count* is initially 0 (*line* 6) and is increased by 1 in each iteration (*line* 39).
T1/AMLT/S7	A subtraction question is displayed and processed in each iteration.
T1/AMLT/S8	The program obtains the time before the test starts in *line* 7 and the time after the test ends in *line* 45, and computes the test time in *line* 46.
T1/AMLT/S9	The test time is in milliseconds and is converted to seconds in *line 50.*
4.2.2	CONTROLLING A *LOOP* WITH A CONFIRMATION DIALOG
T1/LCD/S1	The preceding example executes the *loop* ten times.
T1/LCD/S2	If you want the user to decide whether to take another question, you can use a confirmation dialog to control the *loop*.
T1/LCD/S3	A confirmation dialog can be created using the following *statement*:
T1/LCD/S4	When a button is clicked, the *method* returns an option value.
T1/LCD/S5	The value is *JOptionPane.YES_OPTION (0)* for the Yes button, *JOptionPane.NO_OPTION (1)* for the No button, and *JOptionPane.CANCEL_OPTION (2)* for the Cancel button.
T1/LCD/S6	For example, the following *loop* continues to execute until the user clicks the No or Cancel button.
T1/LCD/S7	You can rewrite *Listing 4.1* using a confirmation dialog to let the user decide whether to continue the next question.
4.2.3	CONTROLLING A *LOOP* WITH A SENTINEL VALUE
T1/LSV/S1	Another common technique *for* controlling a *loop* is to designate a special value when reading and processing a set of values.
T1/LSV/S2	This special input value, known as a *sentinel value*, signifies the end of the *loop*.
T1/LSV/S3	*Listing 4.2* writes a program that reads and calculates the *sum* of an unspecified number of integers.
T1/LSV/S4	The input 0 signifies the end of the input.
T1/LSV/S5	Do you need to declare a new variable for each input value?
T1/LSV/S6	No.
T1/LSV/S7	Just use one variable named data (*line* 9) to store the input value and use a variable named *sum* (*line* 12) to store the total.
T1/LSV/S8	Whenever a value is read, assign it to data and added to *sum* (*line* 14) if it is not zero.
T1/LSV/S9	A sample run of the program is shown in *Figure 4.3*.
T1/LSV/S10	If data is not 0, it is added to the *sum* (*line* 14) and the next items of input data are read (*line*s 12–19).
T1/LSV/S11	If data is 0, the *loop* body is no longer executed and the *while loop* terminates.
T1/LSV/S12	The input value 0 is the sentinel value for this *loop*.
T1/LSV/S13	Note that if the first input read is 0, the *loop* body never executes, and the resulting *sum* is 0.
T1/LSV/S14	The program uses a *while loop* to add an unspecified number of integers.
4.3	THE *DO-WHILE LOOP*
T1/DWL/S1	The *do-while loop* is a variation of the *while loop*.
T1/DWL/S2	Its syntax is given below:
T1/DWL/S3	Its execution flow chart is shown in *Figure 4.4.*
T1/DWL/S4	The *loop* body is executed first.
T1/DWL/S5	Then the *loop-continuation-condition* is evaluated.
T1/DWL/S6	If the evaluation is true, the *loop* body is executed again.

T1/DWL/S7	If it is false, the *do-while loop* terminates.
T1/DWL/S8	The major difference between a *while loop* and a *do-while loop* is the order in which the *loop-continuation-condition* is evaluated and the *loop* body executed.
T1/DWL/S9	The *while loop* and the *do-while loop* have equal expressive power.
T1/DWL/S10	Sometimes one is a more convenient choice than the other.
T1/DWL/S11	For example, you can rewrite the *while loop* in *Listing 4.2* using a *do-while loop*, as shown in *Listing 4.3.*
4.4	THE *FOR LOOP*
T1/FL/S1	Often you write a *loop* in the following common form.
T1/FL/S2	A *for loop* can be used to simplify the preceding *loop*.
T1/FL/S3	In general, the syntax of a *for loop* is as shown below.
T1/FL/S4	The flow chart of the *for loop* is shown in *Figure 4.5(a).*
T1/FL/S5	The *for loop statement* starts with the keyword *for*, followed by a pair of parentheses enclosing *initial-action*, *loop-continuation-condition*, and *action-after-each-iteration*, and followed by the *loop* body enclosed inside braces.
T1/FL/S6	*Initial-action*, *loop-continuation-condition*, and *action-after-each-iteration* are separated by semicolons.
T1/FL/S7	A *for loop* generally uses a variable to control how many times the *loop* body is executed and when the *loop* terminates.
T1/FL/S8	This variable is referred to as a control variable.
T1/FL/S9	The *initial-action* often initializes a control variable, the *action-after-each-iteration* usually increments or decrements the control variable, and the *loop-continuation-condition* tests whether the control variable has reached a termination value.
T1/FL/S10	For example, the following *for loop* prints "Welcome to *Java*!" a hundred times.
T1/FL/S11	The flow chart of the *statement* is shown in *Figure 4.5(b)*.
T1/FL/S12	The *for loop* initializes *i* to 0, then repeatedly executes the *println statement* and evaluates *i++ while i* is less than *100.*
T1/FL/S13	The *initial-action, i=0,* initializes the control variable, *i.*
T1/FL/S14	The *loop-continuation-condition*, *i< 100* is a Boolean expression.
T1/FL/S15	The expression is evaluated at the beginning of each iteration.
T1/FL/S16	If this condition is true, execute the *loop* body.
T1/FL/S17	If it is false, the *loop* terminates and the program control turns to the *line* following the *loop*.
T1/FL/S18	The *action-after-each-iteration, i++,* is a *statement* that adjusts the control variable.
T1/FL/S19	This *statement* is executed after each iteration.
T1/FL/S20	It increments the control variable.
T1/FL/S21	Eventually, the value of the control variable should force the *loop-continuation-condition* to become false.
T1/FL/S22	Otherwise the *loop* is infinite.
T1/FL/S23	If there is only one *statement* in the *loop* body, as in this example, the braces can be omitted.
T1/FL/S24	The control variable must always be declared inside the control structure of the *loop* or before the *loop*.
T1/FL/S25	If the *loop* control variable is used only in the *loop*, and not elsewhere.
T1/FL/S26	It is good programming practice to declare it in the *initial-action* of the *for loop*.
T1/FL/S27	If the variable is declared inside the *loop* control structure, it cannot be referenced outside the *loop*.
T1/FL/S28	For example, you cannot reference *i* outside the *for loop* in the preceding code, because it is declared inside the *for loop*.

4.5.	WHICH *LOOP* TO USE?
T1/WLU/S1	The *while loop* and *for loop* are called pre-test *loops* because the continuation condition is checked be*fo*re the *loop* body is executed.
T1/WLU/S2	The *do-while loop* is called a post-test *loop* because the condition is checked after the *loop* body is executed.
T1/WLU/S3	The three *for*ms of *loop statements*, *while*, *do-while*, and *for*, are expressively equivalent.
T1/WLU/S4	That is, you can write a *loop* in any of these three forms.
T1/WLU/S5	For example, a *while loop* in (a) in the following *Figure* can always be converted into the *for loop* in (b):
T1/WLU/S6	A *for loop* in (a) in the next *Figure* can generally be converted into the *while loop* in (b) except in certain special cases (see Review Question 4.12 *for* such a case):
T1/WLU/S7	Use the *loop statement* that is most intuitive and comfortable for you.
T1/WLU/S8	In general, a *for loop* may be used if the number of repetitions is known, as, for example, when you need to *print* a message a hundred times.
T1/WLU/S9	A *while loop* may be used if the number of repetitions is not known, as in the case of reading the numbers until the input is 0.
T1/WLU/S10	A *do-while loop* can be used to replace a *while loop* if the *loop* body has to be executed before the continuation condition is tested.
4.6.	*NESTED LOOPS*
T1/NL/S1	*Nested loops* consist of an outer *loop* and one or more inner *loops*.
T1/NL/S2	Each time the outer *loop* is repeated, the inner *loops* are reentered, and started anew.
T1/NL/S3	*Listing 4.4* presents a program that uses *nested for loops* to *print* a multiplication table, as shown in *Figure* 4.6.
T1/NL/S4	The program displays a title (*line* 7) on the first *line* and dashes (-) (*line* 8) on the second *line*.
T1/NL/S5	The first *for loop* (*line*s 12–13) displays the numbers 1 through 9 on the third *line*.
T1/NL/S6	The next *loop* (*line*s 18–28) is a *nested for loop* with the control variable *i* in the outer *loop* and *j* in the inner *loop*.
T1/NL/S7	For each i, the product *i * j* is displayed on a *line* in the inner *loop*, with *j* being *1, 2, 3, …, 9.*
T1/NL/S8	The *if statement* in the inner *loop* (*line*s 22–25) is used so that the product will be aligned properly.
T1/NL/S9	If the product is a single digit, it is displayed with an extra space before it.
4.7	MINIMIZING NUMERICAL ERRORS
T1/MNE/S1	Numeric errors involving floating-point numbers are inevitable.
T1/MNE/S2	This section discusses how to minimize such errors through an example.
T1/MNE/S3	*Listing 4.5* presents an example that *sum*s a series that starts with *0.01* and ends with *1.0.*
T1/MNE/S4	The numbers in the series will increment by 0.01, as follows: *0.01 + 0.02 + 0.03* and so on.
T1/MNE/S5	The output of the program appears in *Figure 4.7.*
T1/MNE/S6	The *for loop* (*line*s 9–10) repeatedly adds the control variable *i* to the *sum*.
T1/MNE/S7	This variable, which begins with *0.01*, is incremented by *0.01* after each iteration.
T1/MNE/S8	The *loop* terminates when *i* exceeds *1.0.*
T1/MNE/S9	The *for loop initial action* can be any *statement*, but it is often used to initialize a control variable.
T1/MNE/S10	From this example, you can see that a control variable can be a *float* type.
T1/MNE/S11	In fact, it can be any data type.
T1/MNE/S12	The exact *sum* should be *50.50,* but the answer is *50.499985.*
T1/MNE/S13	The result is not precise because computers use a fixed number of bits to represent floating-point numbers, and thus cannot represent

	some *floating*-point numbers exactly.
T1/MNE/S14	If you change *float* in the program to *double* as follows, you should see a slight improvement in precision because a *double* variable takes sixty-four *bits*, whereas a *float* variable takes thirty-two *bits*.
T1/MNE/S15	However, you will be stunned to see that the result is actually *49.50000000000003.*
T1/MNE/S16	What went wrong?
T1/MNE/S17	If you *print* out *i* for each iteration in the *loop*, you will see that the last *i* is slightly larger than 1 (not exactly 1).
T1/MNE/S18	This causes the last *i* not to be added into *sum*.
T1/MNE/S19	The fundamental problem is that the *floating-point* numbers are represented by approximation.
T1/MNE/S20	Errors commonly occur.
T1/MNE/S21	There are two ways to fix the problem.
T1/MNE/S22	Minimizing errors by processing large numbers first.
T1/MNE/S23	Using an integer *count* to ensure that all the numbers are processed.
T1/MNE/S24	To minimize errors, add numbers from *1.0, 0.99,* down to *0.1,* as follows:
T1/MNE/S25	To ensure that all the items are added to *sum*, use an integer variable to *count* the items.
T1/MNE/S26	Here is the new *loop*.
T1/MNE/S27	After this *loop*, *sum* is *50.50000000000003.*
4.8	CASE STUDIES
T1/CS/S1	Control *statements* are fundamental in programming.
T1/CS/S2	The ability to write control *statements* is essential in learning *Java* programming.
T1/CS/S3	If you can write programs using *loops*, you know how to program!
T1/CS/S4	For this reason, this section presents three additional examples of how to solve problems using *loops*.
4.8.1	EXAMPLE: FINDING THE GREATEST COMMON DIVISOR
T1/FGCD/S1	This section presents a program that prompts the user to enter two positive integers and finds their greatest common divisor.
T1/FGCD/S2	The greatest common divisor of two integers *4* and *2* is *2.*
T1/FGCD/S3	The greatest common divisor of two integers *16* and *24* is *8.*
T1/FGCD/S4	How do you find the greatest common divisor?
T1/FGCD/S5	Let the two input integers be *n1* and *n2.*
T1/FGCD/S6	You know that number 1 is a common divisor, but it may not be the greatest common divisor.
T1/FGCD/S7	So you can check whether *k (*for *k = 2, 3, 4* and so on) is a common divisor for *n1* and *n2*, until *k* is greater than *n1* or *n2.*
T1/FGCD/S8	Store the common divisor in a variable named *gcd.*
T1/FGCD/S9	Initially, *gcd* is *1.*
T1/FGCD/S10	Whenever a new common divisor is found, it becomes the new *gcd.*
T1/FGCD/S11	When you have checked all the possible common divisors from *2* up to *n1* or *n2*, the value in variable *gcd* is the greatest common divisor.
T1/FGCD/S12	The idea can be translated into the following *loop*:
T1/FGCD/S13	The complete program is given in *Listing* 4.6, and a sample run of the program is shown in *Figure 4.8.*
T1/FGCD/S14	The program finds the greatest common divisor for two integers.
T1/FGCD/S15	How did you write this program?

T1/FGCD/S16	Did you immediately begin to write the code?
T1/FGCD/S17	No.
T1/FGCD/S18	It is important to think before you type.
T1/FGCD/S19	Thinking enables you to generate a logical solution for the problem without concern about how to write the code.
T1/FGCD/S20	Once you have a logical solution, type the code to translate the solution into a *Java* program.
T1/FGCD/S21	The translation is not unique.
T1/FGCD/S22	For example, you could use a *for loop* to rewrite the code as follows:
T1/FGCD/S23	A problem often has multiple solutions.
T1/FGCD/S24	The *GCD* problem can be solved in many ways.
T1/FGCD/S25	*Exercise 4.15* suggests another solution.
T1/FGCD/S26	A more efficient solution is to use the classic Euclidean algorithm.
T1/FGCD/S27	See *http://www.cut-the-knot.org/blue/Euclid.shtml* for more in*for*mation.
T1/FGCD/S28	You might think that a divisor for a number *n1* cannot be greater than *n1* / *2.*
T1/FGCD/S29	So you would attempt to improve the program using the following *loop*:
T1/FGCD/S30	This revision is wrong.
T1/FGCD/S31	Can you find the reason?
T1/FGCD/S32	See *Review Question 4.9* for the answer.
4.8.2	EXAMPLE: FINDING THE SALES AMOUNT
T1/FSA/S1	You have just started a sales job in a department store.
T1/FSA/S2	Your pay consists of a base salary and a commission.
T1/FSA/S3	The base salary is *$5,000.*
T1/FSA/S4	The scheme shown below is used to determine the commission rate.
T1/FSA/S5	Your goal is to earn *$30,000* a year.
T1/FSA/S6	This section writes a program that finds the minimum amount of sales you have to generate in order to make *$30,000.*
T1/FSA/S7	Since your base salary is *$5,000*, you have to make *$25,000* in commissions to earn *$30,000* a year.
T1/FSA/S8	What is the sales amount *for* a *$25,000* commission?
T1/FSA/S9	If you know the sales amount, the commission can be computed as follows:
T1/FSA/S10	This suggests that you can try to find the *salesAmount* to match a given commission through incremental approximation.
T1/FSA/S11	For a *salesAmount* of *$0.01 (1 cent),* find commission.
T1/FSA/S12	If commission is less than *$25,000*, increment *salesAmount* by *0.01* and find commission again.
T1/FSA/S13	If commission is still less than *$25,000,* repeat the process until it is greater than or equal to *$25,000.*
T1/FSA/S14	This is a tedious job for humans, but it is exactly what a computer is good for.
T1/FSA/S15	You can write a *loop* and let a computer execute it painlessly.
T1/FSA/S16	The idea can be translated into the following *loop*:
T1/FSA/S17	The complete program is given in *Listing* 4.7, and a sample run of the program is shown in *Figure* 4.9.
T1/FSA/S18	The *do-while loop* (*line*s 12–24) is used to repeatedly compute commission for an incremental *salesAmount.*
T1/FSA/S19	The *loop* terminates when commission is greater than or equal to a constant *COMMISSION_SOUGHT.*

T1/FSA/S20	In *Exercise 4.17*, you will rewrite this program to let the user enter *COMMISSION_SOUGHT* dynamically from an input dialog.
T1/FSA/S21	You can improve the performance of this program by estimating a higher *INITIAL_SALES_AMOUNT (e.g., 25000).*
T1/FSA/S22	What is wrong if *salesAmount* is incremented after the commission is computed, as follows?
T1/FSA/S23	The change is erroneous because *salesAmount* is 1 cent more than is needed for the commission when the *loop* ends.
T1/FSA/S24	This is a common error in *loops*, known as the *off-by-one error*.
4.8.3	EXAMPLE: DISPLAYING A PYRAMID OF NUMBERS
T1/DPN/S1	This section presents a program that prompts the user to enter an integer from *1* to *15* and displays a pyramid.
T1/DPN/S2	If the input integer is *12*, for example, the output is shown in *Figure 4.10.*
T1/DPN/S3	The program uses *nested loops* to *print* numbers in a triangular pattern.
T1/DPN/S4	Your program receives the input *for* an integer *(numberOfLines)* that represents the total number of lines.
T1/DPN/S5	It displays all the lines one by one.
T1/DPN/S6	Each line has three parts.
T1/DPN/S7	The first part comprises the spaces before the numbers;
T1/DPN/S8	The second part, the leading numbers, such as 3 2 1 in *line* 3.
T1/DPN/S9	And the last part, the ending numbers, such as 2 3 in *line* 3.
T1/DPN/S10	Each number occupies three spaces.
T1/DPN/S11	Display an empty space before a double-digit number, and display two empty spaces before a single-digit number.
T1/DPN/S12	You can use an outer *loop* to control the lines.
T1/DPN/S13	At the n^{th} row, there are (*numberOfLines – n)*3* leading spaces, the leading numbers are *n, n-1, … 1,* and the ending numbers are *2, ..., n.*
T1/DPN/S14	You can use three separate inner *loops* to *print* each part.
T1/DPN/S15	Here is the algorithm for the problem
T1/DPN/S16	The complete program is given in *Listing 4.8.*
T1/DPN/S17	The program uses the *print method (lines 20, 24, and 28)* to display a *string* to the console.
T1/DPN/S18	The conditional expression *(num >= 10) ? " " + num : " " + num* in *line*s 24 and 28 returns a *string* with a single empty space before the number if the number is greater than or equal to 10, and otherwise returns a *string* with two empty spaces before the number.
T1/DPN/S19	Printing patterns like this one and the ones in Exercises 4.18 and 4.19 is a good exercise for practicing *loop* control *statements*.
T1/DPN/S20	The key is to understand the pattern and to describe it using *loop* control variables.
T1/DPN/S21	The last line in the outer *loop* (*line* 31), *System.out.println(),* does not have any argument in the *method*.
T1/DPN/S22	This call moves the cursor to the next line.
4.9	KEYWORDS BREAK AND CONTINUE
T1/KBC/S1	Two *statements*, *break* and *continue*, can be used in *loop statements* to provide the *loop* with additional control.
T1/KBC/S2	*break* immediately ends the innermost *loop* that contains it.
T1/KBC/S3	It is generally used with an *if statement*.
T1/KBC/S4	*continue* only ends the current iteration.
T1/KBC/S5	Program control goes to the end of the *loop* body.
T1/KBC/S6	This keyword is generally used with an *if statement*.

T1/KBC/S7	You have already used the keyword *break* in a *switch statement*.
T1/KBC/S8	You can also use *break* and *continue* in a *loop*.
T1/KBC/S9	*Listings 4.9* and *4.10* present two programs to demonstrate the effect of the *break* and *continue* keywords in a *loop*.
T1/KBC/S10	The program in *Listing 4.9* adds the integers from *1 to 20* in this order to *sum* until *sum* is greater than or equal to *100.*
T1/KBC/S11	Without the *if statement* (*line* 10), the program calculates the *sum* of the numbers from *1 to 20.*
T1/KBC/S12	But with the *if statement*, the *loop* terminates when the *sum* becomes greater than or equal to *100.*
T1/KBC/S13	The output of the program is shown in *Figure 4.11(a).*
T1/KBC/S14	If you changed the *if statement* as shown below, the output would resemble that in *Figure 4.11(b).*
T1/KBC/S15	In this case, the *if* condition will never be true.
T1/KBC/S16	Therefore, the *break statement* will never be executed.
T1/KBC/S17	The program in *Listing 4.10* adds all the integers from *1 to 20* except *10 and 11* to *sum*.
T1/KBC/S18	With the *if statement* in the program (*line* 9), the *continue statement* is executed when number becomes *10 or 11*.
T1/KBC/S19	The *continue statement* ends the current iteration so that the rest of the *statement* in the *loop* body is not executed; therefore, number is not added to *sum* when it is 10 or 11.
T1/KBC/S20	The output of the program is shown in *Figure 4.12(a).*
T1/KBC/S21	Without the if *statement* in the program, the output would look like *Figure 4.12(b).*
T1/KBC/S22	In this case, all of the numbers are added to *sum*, even when number is *10 or 11.*
T1/KBC/S23	Therefore, the result is *210*, which is *21* more than it was with the *if statement*.

Appendix 2B

Text 2 – Coding of Sentences

CHAPTER 9	FEELING A LITTLE LOOPY
TITLE	A *LOOP* FOR EVERY OCCASION (LEO)
T2/LEO/S1	Have you ever been talking to someone and it seems like he or she is saying the same thing over and over?
T2/LEO/S2	I mean, you keep listening, and they keep talking, and it all sounds the same.
T2/LEO/S3	And they talk somemore and you listen somemore and you wonder if it will ever end!
T2/LEO/S4	Congratulations, you just experienced a perfect example of a verbal *loop*!
T2/LEO/S5	In *Java*, a *loop* is a programming construct that enables you to repeat a section of code over and over, much like my conversation example.
T2/LEO/S6	*Loops* are very valuable in *Java* because they enable you to tightly control repetitive functions.
T2/LEO/S7	Three type of *loops* are used in *Java*: *for loops*, *while loops* and *do loops*.
TITLE	GETTING REDUNDANT WITH THE *FOR LOOP* (GRL)
T2/GRL/S1	Let's pretend NASA used *Java* applets to control the launch of the space shuttle.
T2/GRL/S2	Any ideas on how controllers would initiate the launch sequence?
T2/GRL/S3	With *loops*!
T2/GRL/S4	Counting down from ten to one is a piece of cake with a *loop*.
T2/GRL/S5	Granted, without a *loop* it wouldn't be too tough either, but it would require some unnecessary code.
T2/GRL/S6	Following is code to perform the launch sequence without the use of a *loop*.
T2/GRL/S7	And now the *loop* version:
T2/GRL/S8	See what I mean about tightening up the code?
T2/GRL/S9	You probably wonder exactly how the *loop* code works.
T2/GRL/S10	This code relies on a *for loop*, which is the most structured type of *loop* supported by *Java*.
T2/GRL/S11	For *loops* repeat a section of code a fixed number of times.
T2/GRL/S12	Following is the syntax *for* the *for loop*:
T2/GRL/S13	The *for loop* repeats the *Statement* the number of time determined by the *InitializationExpression, LoopCondition* and *StepExpression:*
T2/GRL/S14	The *InitializationExpression* is used to initialize a *loop* control variable.
T2/GRL/S15	The *LoopCondition* compares the *loop* control variable to some limit or value.
T2/GRL/S16	The *StepExpression* specifies how the *loop* control variable should be modified before the next iteration of the *loop*.
T2/GRL/S17	Let's take a look at the NASA launch sequence code again to make some sense of this stuff.
T2/GRL/S18	In this code the *InitializationExpression* is *int i310*, which is evaluated initially before the *loop* begins.
T2/GRL/S19	This is the code you use to prime the *loop* and get it ready.
T2/GRL/S20	The *LoopCondition* is *i>0*, which is a *Boolean* test that is performed before each iteration of the *loop*.

T2/GRL/S21	If the *Boolean* test result is true, the *Statement* is executed, which in this case prints the current value of *i.*
T2/GRL/S22	After each iteration the *StepExpression* is evaluated, which is *i--.*
T2/GRL/S23	This serves to decrement *i* after each iteration, and ultimately proves the countdown.
T2/GRL/S24	The *loop* continues to iterate and print numbers as *i* counts down to 0.
T2/GRL/S25	After i reaches 0, the *LoopCondition* test fails (i>0), so the *loop* bails out without printing any more numbers.
T2/GRL/S26	Whew, that explanation seemed a little long-winded, and that's coming from the person that wrote it!
T2/GRL/S27	Unfortunately, it isn't always easy to verbalize the flow of program code.
T2/GRL/S28	This is why it's easy to fall back on figures.
T2/GRL/S29	Just ask Ross Perot, who isn't a *Java* programmer but who nonetheless relied on diagrams and illustrations to help us grasp his big plans for the presidency.
T2/GRL/S30	You can feel safe and secure knowing that I'm not running *for* president or trying to help you visualize my answer to global trade.
T2/GRL/S31	I just want to help you learn how *loops* work!
T2/GRL/S32	To help you visualize the *looping* process, take a look at the following figure.
T2/GRL/S33	Notice in the figure that *Statement 1* and *Statement 2* will be repeatedly executed as long as the *loop* condition is true.
T2/GRL/S34	When the *loop* condition goes false, the program falls out of the *loop* and executes *Statement 3.*
T2/GRL/S35	The previous figure alludes to the fact that a *loop* can execute multiple *statements*.
T2/GRL/S36	*Loops* can execute as many *statements* as they want, provided curly braces ({}) enclose the *statements.*
T2/GRL/S37	If you recall, this grouping of *statements* is known as a *compound statement* and was used in the previous chapter when dealing with *if-else* branches.
T2/GRL/S38	Following is an example of a *for loop* with a *compound statement:*
T2/GRL/S39	This code calculates the squares of the numbers 1 through 10, stores them in an array, and prints each one.
T2/GRL/S40	Notice that the *loop counter (i)* is used as the index *(i-1)* into the squares array.
T2/GRL/S41	This is a very popular way to handle arrays.
T2/GRL/S42	It is necessary to subtract 1 in this case because all *Java* array indexes start with 0, which means they are zero based.
T2/GRL/S43	It might be worth nothing that although zero-based arrays were used in other programming languages in the 1980s and before, they have nothing to *do* with the 80s movie Less than Zero or the 80s hit song saved by Zero.
T2/GRL/S44	Rest assured I would be the first to tell you if they did!
TITLE	*LOOP*ING FOR JUST A LITTLE WHILE (LJLW)
T2/LJLW/S1	Like the *for loop*, the *while loop* has a *loop* condition that controls the number of times a *loop* is repeated.
T2/LJLW/S2	However, the *while loop* has no *initialization* or *step expression.*
T2/LJLW/S3	A *for loop* is like one of those friends who tells you a story three or four times and then waits for a response, whereas a *while loop* is like one of those friends who continues to repeat himself as long as you continue to listen.
T2/LJLW/S4	They're both annoying, but in different ways.
T2/LJLW/S5	Not the *loops*, the people!
T2/LJLW/S6	Following is the syntax *for* the *while loop*, which should make its usage a little more clear:
T2/LJLW/S7	If the *Boolean LoopCondition* evaluates to true, the *Statement* is executed.
T2/LJLW/S8	When the *Statement* finishes executing, the *LoopCondition* is tested again and the process repeats itself.

T2/LJLW/S9	This continues until the *LoopCondition* evaluates to false, in which case the *loop* immediately bails out.
T2/LJLW/S10	Because the *while loop* has no *step expression*, it is important to make sure that the *Statement* somehow impacts the *LoopCondition*.
T2/LJLW/S11	Otherwise, it is possible *for* the *loop* to repeat infinitely, which is usually a bad thing.
T2/LJLW/S12	Following is a simple example of an infinite *while loop*:
T2/LJLW/S13	Because the *loop* condition in this example is permanently set to true, the *loop* will repeat infinitely, or at least until you manually terminate the programme.
T2/LJLW/S14	Infinite *loops* are extremely dangerous because they can result in your computer overheating.
T2/LJLW/S15	Just kidding!
T2/LJLW/S16	Actually, infinite *loops* are useful in some situations.
T2/LJLW/S17	They are never truly infinite because you can typically terminate one by shutting down the application or applet containing it.
T2/LJLW/S18	You can think of the *while loop* as a more general *for loop*.
T2/LJLW/S19	To understand what I mean by this, check out the following code.
T2/LJLW/S20	This is the NASA launch sequence implemented using a *while loop* instead of a *for loop*.
T2/LJLW/S21	Because *while loops* don't have *initialization* expressions, the *initialization* of the *counter* variable I had to be performed before the *loop*.
T2/LJLW/S22	Likewise, the *step expression i* had to be performed within the *Statement* part of the *loop*.
T2/LJLW/S23	Regardless of the structural differences, this *while loop* is functionally equivalent to the *for loop* you saw earlier in the chapter.
T2/LJLW/S24	If a *for loop* can do everything a *while loop* can and in a more organized way, then why do we need *while loops*?
T2/LJLW/S25	Because there is a time and a place for everything, and in many situations you have no need *for initialization* and *step expressions.*
T2/LJLW/S26	A *for loop* is overkill in situations like this.
T2/LJLW/S27	Even more importantly, a *while loop* is much more readable than a *for loop* when you have no need *for initialization* and *step expressions.*
T2/LJLW/S28	Consider the following example:
T2/LJLW/S29	This code demonstrates how a *while loop* could be used to ask a question and patiently wait for the correct answer.
T2/LJLW/S30	The *loop* repeats itself as long as the *Boolean* variable correct is false.
T2/LJLW/S31	This results in the code repeating the question as many times as necessary until the user guesses the correct answer.
T2/LJLW/S32	The details of the methods *askQuestion()* and *isCorrect()* aren't important for this example.
T2/LJLW/S33	Just assume that they somehow present the user with a question, retrieve an answer, and then judge the correctness of the answer.
T2/LJLW/S34	The main concern is that the *isCorrect () method* returns a *Boolean* value that indicates whether or not the answer is correct.
T2/LJLW/S35	In this example, it is impossible to know how many times the user will miss the answer and need the question repeated.
T2/LJLW/S36	For this reason, the structured *step expression* of a *for loop* wouldn't be of much use.
T2/LJLW/S37	While *loops* are perfect in situations where you don't know ahead of time how many times a *loop* needs to be repeated.
T2/LJLW/S38	If you aren't completely satisfied with *while loops*, however, there is one other option.
TITLE	TO *DO* , OR NOT TO *DO* (TDNTD)
T2/TDNTD/S1	The *while loop* has a very close relative known as the *do loop*, or *do-while loop*, that is surprisingly similar to the *while loop*.
T2/TDNTD/S2	Because you're becoming pretty *loop* savvy, I'll show you the syntax *for* the *do-while loop* first and see if you can figure out how it works.

T2/TDNTD/S3	Give up?
T2/TDNTD/S4	The *do-while loop* is basically a *while loop* with the *LoopCondition* moved to the end.
T2/TDNTD/S5	Why is this necessary?
T2/TDNTD/S6	Because there are some situations where you would like the *Statement* to execute before evaluating the *LoopCondition*, instead of afterward.
T2/TDNTD/S7	This also guarantees that the *Statement* is executed at least once, regardless of the *LoopCondition*.
T2/TDNTD/S8	Let's take a look at the question and answer example implemented using a *do-while loop*:
T2/TDNTD/S9	The code really isn't much different than before, except that you no longer need to *initialise* the correct variable.
T2/TDNTD/S10	It is always initially set during the first pass through the *loop*.
T2/TDNTD/S11	Although both types of *loops* accomplish the goal of this example, the *do-while loop* is a better fit because of its structure more closely mimics the function of the code.
T2/TDNTD/S12	What do I mean by this?
T2/TDNTD/S13	Well, if you "read" the code, it is saying "ask the question and if the answer is not correct, ask it again."
T2/TDNTD/S14	This makes more sense than if it read "if the answer is not correct, ask the question and then check the answer again."
T2/TDNTD/S15	Admittedly, this is a subtle difference, but a large part of successful programming is keeping things logical and straightforward.
T2/TDNTD/S16	You won't always succeed because sometimes code gets complicated regardless of how you construct it, but using *loops* intelligently is a good start.
TITLE	APPLET COUNTDOWN (AC)
T2/AC/S1	Have you ever visited a Web page that directed you to another page, but informed you that if you waited a few second sit would automatically take you there?
T2/AC/S2	I used to run across these pages and wonder how you could make a page wait a few seconds and then automatically navigate to a new page.
T2/AC/S3	After I started programming in *Java*, I realised what a trivial task this is.
T2/AC/S4	In this section you use your knowledge of *loops* to build a "countdown" applet that counts down from ten to one and then navigates to a new Web page.
T2/AC/S5	The following figure shows the *Countdown* applet in action.
T2/AC/S6	When the applet finishes counting down, it navigates to the web page identified by the page applet parameter.
T2/AC/S7	As an example, what web site could be better than NASA's to demonstrate how this applet works?
T2/AC/S8	Following is NASA's Web site, to which the *Countdown* applet will take you after it finishes its countdown.
T2/AC/S9	To understand how the *Countdown* applet works, let's first take a look at the *Countdown. Html* Web page that contains the embedded applet.
T2/AC/S10	All this stuff should look pretty familiar to you by now.

T2/AC/S11	The main thing on which I want you to focus is the page parameter, which is defined as.
T2/AC/S12	Notice that the value of the page parameter is set to http://www.nasa.gov, which is the URL of NASA's Web site.
T2/AC/S13	Changing this value enables you to change the page that is loaded after the applet finishes counting down.
T2/AC/S14	This page could have easily been set as a variable within the applet code, but a recompile would be required to change the page.
T2/AC/S15	That is the beauty of applet parameters.
T2/AC/S16	They enable you to customize the function of applets without doing any real programming!
T2/AC/S17	Let's move on to the actual code required of the countdown applet.
T2/AC/S18	Unfortunately, the *Countdown* applet requires some code that is a little beyond the lesson, so I don't expect all of this applet to make sense to you.
T2/AC/S19	However, you can download the complete source code for the applet from the book's companion Web site, which was mentioned a little earlier in this section.
T2/AC/S20	Also, the core mechanics of the applet are very straightforward and should be familiar to you from your recent study of *loops*.
T2/AC/S21	Following is the *run() method* in the *Countdown* applet class, which forms the heart of the applet:
T2/AC/S22	Ouch, that looks a little messy!
T2/AC/S23	Try not to get intimidated by any code that doesn't look familiar.
T2/AC/S24	Just concentrate on the *loop* code.
T2/AC/S25	As you can see, the *for loop* counts down from 10 to 1 just like the *Countdown* code you saw earlier in the chapter.
T2/AC/S26	The *Statement* part of this *for loop* is completely new territory, however.
T2/AC/S27	The call to the *repaint () method* is necessary to update the applet's window with the new *countdown* number.
T2/AC/S28	The call to the *thread.sleep() method* results in the applet waiting one second, which effectively pauses the *countdown* for one second between numbers.
T2/AC/S29	When the *for loop* finishes, the code gets the page applet parameter and proceed to navigate to the Web page identified by it.
T2/AC/S30	The code required to navigate to the Web page is probably pretty strange looking to you because it has to deal with *exceptions.*
T2/AC/S31	*Exceptions* are errors caused by unforeseen problems such as your computer running out of memory, your modem coming unplugged, spilling coffee on your keyboard, hurling your monitor out the window, and so on.
T2/AC/S32	I'll explain *exceptions* as you encounter them throughout the book.
T2/AC/S33	The complete source code for the *countdow*n applet follows.
T2/AC/S34	Although this is a longer program than you are accustomed to seeing, a lot of it should look familiar to you.
T2/AC/S35	For example, the *paint() method* code is very similar to the code used in the *DateTime* applet from Chapter 4, "constructing Applets of Your Own."
T2/AC/S36	On the other hand, the *start() method* is entirely new and is related to the applet's use of *threads.*
T2/AC/S37	You don't need to understand it fully at this point.
TITLE	BREAKING AWAY (BA)
T2/BA/S1	If you recall from the previous chapter, each case section of a *switch* branch ends with a *break statement*.
T2/BA/S2	Following is an example to recap:
T2/BA/S3	The purpose of the *break statement* in this example is to bail out of the *switch* branch so that no other code is executed.
T2/BA/S4	The *break statement* serves a similar purpose in *loops*.

T2/BA/S5	It breaks out of a *loop* regardless of the *loop* condition.
T2/BA/S6	Following is an example of circumventing an infinite *loop* with a *break statement*:
T2/BA/S7	Without the assistance of the *break statement*, this *while loop* would continue forever thanks to the permanent *true loop* condition.
T2/BA/S8	The *break statement* sidesteps this problem by breaking out of the *loop* after one hundred iterations (0-99).
T2/BA/S9	Of course, it is rare that you would purposely create an infinite *loop* and then use a *break statement* to bail out of it.
T2/BA/S10	However, the *break statement* can be very useful in some tricky *loops* when you need to exit at an otherwise inconvenient time.
T2/BA/S11	A close relative of the *break statement* is the *continue statement*, which is used to skip to the next iteration of a *loop*.
T2/BA/S13	The following example shows how a *continue statement* can be used to *print* only the even numbers between 1 and 100:
T2/BA/S14	Having trouble seeing how this one works?
T2/BA/S15	Think back to the *modulus operator* (%), which returns the remainder of a division.
T2/BA/S16	Now consider what the remainder of a division by 2 yields for even and odd numbers.
T2/BA/S17	Aha!
T2/BA/S18	Even numbers divided by 2 always yield a remainder of 0, and odd numbers always leave a remainder of 1!
T2/BA/S19	The example code exploits this characteristic of even and odd numbers to skip to the next iteration bypasses the *println()* call, which prevents odd numbers from being printed.
T2/BA/S20	Pretty tricky!
TITLE	THE LEAST YOU NEED TO KNOW (LNK)
T2/LNK/S1	Computers are often called upon to perform tasks we humans find to be utterly redundant.
T2/LNK/S2	As dull as some humans can be, I guarantee you computers are much duller when it comes to repeating the same thing over and over.
T2/LNK/S3	*Java* enables you to build programs that repeat themselves through the use of *loops*.
T2/LNK/S4	The different types of *loops* basically perform the same function.
T2/LNK/S5	They repeat a section of code over and over.
T2/LNK/S6	Let's go over the main points you learned about *loops* in this chapter.
T2/LNK/S7	*Loops* can execute as many *statements* as you want them to, provided the *statements* are grouped together as a single *compound statement* enclosed by curly braces ({}).
T2/LNK/S8	A *for loop* is used to repeat a section of code a given number of iterations.
T2/LNK/S9	A *while loop* is a more general *for loop*.
T2/LNK/S10	A *do while* is a *while loop* with the *loop* condition moved to the end.
T2/LNK/S11	The *break statement* is used to break out of a *loop* regardless of the *loop* condition.
T2/LNK/S12	The *continue statement* is used to skip to the next iteration of a *loop*.

Appendix 3A

Text 1 – Features of Metaphor of Mood

Note:
1. For every table entry with metaphorical clauses with Metaphor of Mood, the metaphorical sentence precedes the congruent sentence.

Keys:
1. Dec/St/Co denotes Declarative clauses realizing Statement and Command
2. Int/Qu/St denotes Interrogative clauses realizing Question and Statement
3. Int/Qu/Co denotes Interrogative clauses realizing Question and Command

4.1	INTRODUCTION	Code
T1/INT/S1	Suppose that you need to *print* a *string* (e.g., "Welcome to *Java*!") a hundred times.	
T1/INT/S2	It would be tedious to have to write the following *statement* a hundred times.	
T1/INT/S3	*Java* provides a powerful control structure called a *loop* that controls how many times an operation or a sequence of operations is performed in succession.	
T1/INT/S4	Using a *loop statement*, you simply tell the computer to *print* a *string* a hundred times without having to code the *print statement* a hundred times. Using a *loop statement*, tell the computer to *print* a *string* a hundred times without having to code the *print statement* a hundred times.	Dec/St/Co
T1/INT/S5	*Loops* are structures that control repeated executions of a block of *statements*.	
T1/INT/S6	The concept of *looping* is fundamental to programming.	
T1/INT/S7	*Java* provides three types of *loop statements*: *while loops*, *do-while loops*, and *for loops*.	
4.2	THE *WHILE LOOP*	
T1/WL/S1	The syntax *for* the *while loop* is as follows.	
T1/WL/S2	The *while loop* flow chart is shown in *Figure 4.1(a).*	
T1/WL/S3	The part of the *loop* that contains the *statements* to be repeated is called the *loop body*.	
T1/WL/S4	A one-time execution of a *loop* body is referred to as an iteration of the *loop*.	
T1/WL/S5	Each *loop* contains a *loop-continuation* condition, a Boolean expression that controls the execution of the body.	
T1/WL/S6	It is always evaluated be*fore* the *loop* body is executed.	
T1/WL/S7	If its evaluation is true, the *loop* body is executed.	
T1/WL/S8	If its evaluation is false, the entire *loop* terminates and the program control turns to the *statement* that follows the *while loop*.	
T1/WL/S9	For example, the following *while loop* prints "Welcome to *Java*!" a hundred times.	

T1/WL/S10	The flow chart of the preceding *statement* is shown in *Figure 4.1(b).*	
T1/WL/S11	The variable *count* is initially 0.	
T1/WL/S12	The *loop* checks whether (*count* < *100*) is true.	
T1/WL/S13	If so, it executes the *loop* body to *print* the message "Welcome to *Java*!" and increments *count* by 1.	
T1/WL/S14	It repeatedly executes the *loop* body until (*count* < *100*) becomes false.	
T1/WL/S15	When (*count* < *100*) is false (i.e., when *count* reaches *100*), the *loop* terminates and the next *statement* after the *loop statement* is executed.	
4.2.1	AN ADVANCED MATH LEARNING TOOL	
T1/AMLT/S1	The Math subtraction learning tool program in *Listing* 3.5, *SubtractionTutor.Java*, generates just one question for each run.	
T1/AMLT/S2	You can use a *loop* to generate questions repeatedly. Use a *loop* to generate questions repeatedly.	
T1/AMLT/S3	*Listing* 4.1 gives a program that generates ten questions and reports the number of correct answers after a student answers all ten questions.	
T1/AMLT/S4	The program also displays the time spent on the test and lists all the questions, as shown in *Figure 4.2.*	
T1/AMLT/S5	The program uses the control variable *count* to control the execution of the *loop*.	
T1/AMLT/S6	*Count* is initially 0 (*line* 6) and is increased by 1 in each iteration (*line* 39).	
T1/AMLT/S7	A subtraction question is displayed and processed in each iteration.	
T1/AMLT/S8	The program obtains the time before the test starts in *line* 7 and the time after the test ends in *line* 45, and computes the test time in *line* 46.	
T1/AMLT/S9	The test time is in milliseconds and is converted to seconds in *line 50.*	
4.2.2	CONTROLLING A *LOOP* WITH A CONFIRMATION DIALOG	
T1/LCD/S1	The preceding example executes the *loop* ten times.	
T1/LCD/S2	If you want the user to decide whether to take another question, you can use a confirmation dialog to control the *loop*. If you want the user to decide whether to take another question, use a confirmation dialog to control the *loop*.	Dec/St/Co
T1/LCD/S3	A confirmation dialog can be created using the following *statement*. Create a confirmation dialog using the following *statement.*	Dec/St/Co
T1/LCD/S4	When a button is clicked, the *method* returns an option value.	
T1/LCD/S5	The value is *JOptionPane.YES_OPTION (0)* for the Yes button, *JOptionPane.NO_OPTION (1)* for the No button, and *JOptionPane.CANCEL_OPTION (2)* for the Cancel button.	
T1/LCD/S6	For example, the following *loop* continues to execute until the user clicks the *No* or *Cancel* button.	
T1/LCD/S7	You can rewrite *Listing 4.1* using a confirmation dialog to let the user decide whether to continue the next question. Rewrite *Listing 4.1* using a confirmation dialog to let the user decide whether to continue the next question.	Dec/St/Co
4.2.3	CONTROLLING A *LOOP* WITH A SENTINEL VALUE	
T1/LSV/S1	Another common technique *for* controlling a *loop* is to designate a special value when reading and processing a set of values.	
T1/LSV/S2	This special input value, known as a *sentinel value*, signifies the end of the *loop*.	
T1/LSV/S3	*Listing 4.2* writes a program that reads and calculates the *sum* of an unspecified number of integers.	

T1/LSV/S4	The input 0 signifies the end of the input.	
T1/LSV/S5	Do you need to declare a new variable for each input value?	Int/Qu/St
T1/LSV/S6	No. There is no need to declare a new variable for each input.	
T1/LSV/S7	Just use one variable named *data* (*line* 9) to store the input value and use a variable named *sum* (*line* 12) to store the total. One variable named *data* (*line* 9) is used to store the input value and a variable named *sum* (*line* 12) is used to store the total	Imp/Co/St
T1/LSV/S8	Whenever a value is read, assign it to *data* and added to *sum* (*line* 14) if it is not zero.	
T1/LSV/S9	A sample run of the program is shown in *Figure 4.3.* See *Figure 4.3* for a sample run of the program	Dec/St/Co
T1/LSV/S10	If *data* is not 0, it is added to the *sum* (*line* 14) and the next items of input data are read (*line*s 12–19).	
T1/LSV/S11	If data is 0, the *loop* body is no longer executed and the *while loop* terminates.	
T1/LSV/S12	The input value 0 is the sentinel value for this *loop*.	
T1/LSV/S13	Note that if the first input read is 0, the *loop* body never executes, and the resulting *sum* is 0. If the first input read is 0, the *loop* body never executes, and the resulting *sum* is 0.	Imp/Co/St
T1/LSV/S14	The program uses a *while loop* to add an unspecified number of integers.	
4.3	THE *DO-WHILE LOOP*	
T1/DWL/S1	The *do-while loop* is a variation of the *while loop*.	
T1/DWL/S2	Its syntax is given below: See below for its syntax:	Dec/St/Co
T1/DWL/S3	Its execution flow chart is shown in *Figure 4.4.* See *Figure 4.4* for its execution flow chart.	Dec/St/Co
T1/DWL/S4	The *loop* body is executed first.	
T1/DWL/S5	Then the *loop-continuation-condition* is evaluated.	
T1/DWL/S6	If the evaluation is true, the *loop* body is executed again.	
T1/DWL/S7	If it is false, the *do-while loop* terminates.	
T1/DWL/S8	The major difference between a *while loop* and a *do-while loop* is the order in which the *loop-continuation-condition* is evaluated and the *loop* body executed.	
T1/DWL/S9	The *while loop* and the *do-while loop* have equal expressive power.	
T1/DWL/S10	Sometimes one is a more convenient choice than the other.	
T1/DWL/S11	For example, you can rewrite the *while loop* in *Listing 4.2* using a *do-while loop*, as shown in *Listing 4.3.* For example, rewrite the *while loop* in *Listing 4.2* using a *do-while loop*, as shown in *Listing 4.3.*	Dec/St/Co
4.4	THE *FOR LOOP*	
T1/FL/S1	Often you write a *loop* in the following common form.	
T1/FL/S2	A *for loop* can be used to simplify the preceding *loop*. Use a *for loop* to simplify the preceding *loop*.	Dec/St/Co

T1/FL/S3	In general, the syntax of a *for loop* is as shown below. In general, see below for the syntax of a *for loop.*	Dec/St/Co
T1/FL/S4	The flow chart of the *for loop* is shown in *Figure 4.5(a).* See *Figure 4.5(a)* for the flow chart of the *for loop.*	
T1/FL/S5	The *for loop statement* starts with the keyword *for*, followed by a pair of parentheses enclosing *initial-action*, *loop-continuation-condition*, and *action-after-each-iteration*, and followed by the *loop* body enclosed inside braces. Start the *for loop statement* with the keyword *for,* enclose *initial-action, loop-continuation-condition* with a pair of parenthesis, and enclose the *loop* body inside braces.	Dec/St/Co
T1/FL/S6	*Initial-action*, *loop-continuation-condition*, and *action-after-each-iteration* are separated by semicolons. Separate *Initial-action*, *loop-continuation-condition*, and *action-after-each-iteration* with semicolons.	Dec/St/Co
T1/FL/S7	A *for loop* generally uses a variable to control how many times the *loop* body is executed and when the *loop* terminates. Use a variable in a *for loop* to control how many times the *loop* body is executed and when the *loop* terminates.	Dec/St/Co
T1/FL/S8	This variable is referred to as a control variable. Refer this variable as a control variable.	Dec/St/Co
T1/FL/S9	The *initial-action* often initializes a control variable, the *action-after-each-iteration* usually increments or decrements the control variable, and the *loop-continuation-condition* tests whether the control variable has reached a termination value.	
T1/FL/S10	For example, the following *for loop* prints "Welcome to *Java*!" a hundred times. See the following to understand how *for loop* prints "Welcome to *Java*!" a hundred times.	Dec/St/Co
T1/FL/S11	The flow chart of the *statement* is shown in *Figure 4.5(b)*. See *Figure 4.5(b)* for the flow chart of the *statement.*	Dec/St/Co
T1/FL/S12	The *for loop* initializes *i* to 0, then repeatedly executes the *println statement* and evaluates *i++ while i* is less than *100.*	
T1/FL/S13	The *initial-action, i=0,* initializes the control variable, *i.*	
T1/FL/S14	The *loop-continuation-condition*, *i< 100* is a Boolean expression.	
T1/FL/S15	The expression is evaluated at the beginning of each iteration.	
T1/FL/S16	If this condition is true, execute the *loop* body.	
T1/FL/S17	If it is false, the *loop* terminates and the program control turns to the *line* following the *loop*.	
T1/FL/S18	The *action-after-each-iteration, i++,* is a *statement* that adjusts the control variable.	
T1/FL/S19	This *statement* is executed after each iteration.	
T1/FL/S20	It increments the control variable.	
T1/FL/S21	Eventually, the value of the control variable should force the *loop-continuation-condition* to become false.	
T1/FL/S22	Otherwise the *loop* is infinite.	
T1/FL/S23	If there is only one *statement* in the *loop* body, as in this example, the braces can be omitted. If there is only one *statement* in the *loop* body, as in this example, omit the braces.	Dec/St/Co
T1/FL/S24	The control variable must always be declared inside the control structure of the *loop* or before the *loop*. Declare the control variable inside the control structure of the *loop* or before the *loop*.	Dec/St/Co
T1/FL/S25	The *loop* control variable is used only in the *loop*, and not elsewhere. Use the *loop* control variable only in the *loop*, and not elsewhere.	Dec/St/Co

T1/FL/S26	It is good programming practice to declare it in the *initial-action* of the *for loop*.	
T1/FL/S27	If the variable is declared inside the *loop* control structure, it cannot be referenced outside the *loop*.	
T1/FL/S28	For example, you cannot reference *i* outside the *for loop* in the preceding code, because it is declared inside the *for loop*. Do not reference *i* outside the *for loop* in the preceding code, because it is declared inside the *for loop*.	Dec/St/Co
4.5.	WHICH *LOOP* TO USE?	
T1/WLU/S1	The *while loop* and *for loop* are called pre-test *loops* because the continuation condition is checked be*fore* the *loop* body is executed. Refer the *while loop* and *for loop as* pre-test *loops* because the continuation condition is checked be*fore* the *loop* body is executed.	Dec/St/Co
T1/WLU/S2	The *do-while loop* is called a post-test *loop* because the condition is checked after the *loop* body is executed. Refer The *do-while loop as* a post-test *loop* because the condition is checked after the *loop* body is executed	Dec/St/Co
T1/WLU/S3	The three *for*ms of *loop statements*, *while*, *do-while*, and *for*, are expressively equivalent.	
T1/WLU/S4	That is, you can write a *loop* in any of these three forms. Write a *loop* in any of these three forms.	Dec/St/Co
T1/WLU/S5	For example, a *while loop* in (a) in the following figure can always be converted into the *for loop* in (b): See the following figure on how , a *while loop* in (a) can always be converted into the *for loop* in (b):	Dec/St/Co
T1/WLU/S6	A *for loop* in (a) in the next figure can generally be converted into the *while loop* in (b) except in certain special cases (see Review Question 4.12 *for* such a case): See the next figure on how A *for loop* in (a) can generally be converted into the *while loop* in (b) except in certain special cases (see Review Question 4.12 *for* such a case):	Dec/St/Co
T1/WLU/S7	Use the *loop statement* that is most intuitive and comfortable for you. You can use the *loop statement* that is most intuitive and comfortable for you.	Imp/Co/St
T1/WLU/S8	In general, a *for loop* may be used if the number of repetitions is known, as, for example, when you need to *print* a message a hundred times. In general, use a *for loop* used if the number of repetitions is known, as, for example, when you need to *print* a message a hundred times.	Dec/St/Co
T1/WLU/S9	A *while loop* may be used if the number of repetitions is not known, as in the case of reading the numbers until the input is 0. Use a *while loop* if the number of repetitions is not known, as in the case of reading the numbers until the input is 0.	Dec/St/Co
T1/WLU/S10	A *do-while loop* can be used to replace a *while loop* if the *loop* body has to be executed before the continuation condition is tested. Use a *do-while loop* to replace a *while loop* if the *loop* body has to be executed before the continuation condition is tested.	Dec/St/Co
4.6.	*NESTED LOOPS*	
T1/NL/S1	*Nested loops* consist of an outer *loop* and one or more inner *loops*.	
T1/NL/S2	Each time the outer *loop* is repeated, the inner *loops* are reentered, and started anew.	
T1/NL/S3	*Listing 4.4* presents a program that uses *nested for loops* to *print* a multiplication table, as shown in *Figure 4.6*. See Listing 4.4 that presents a program that uses *nested for loops* to *print* a multiplication table, as shown in *Figure 4.6*	Dec/St/Co
T1/NL/S4	The program displays a title (*line* 7) on the first *line* and dashes (-) (*line* 8) on the second *line*.	

T1/NL/S5	The first *for loop* (*line*s 12–13) displays the numbers 1 through 9 on the third *line*.	
T1/NL/S6	The next *loop* (*line*s 18–28) is a *nested for loop* with the control variable *i* in the outer *loop* and *j* in the inner *loop*.	
T1/NL/S7	For each *i,* the product *i * j* is displayed on a *line* in the inner *loop*, with *j* being *1, 2, 3, …, 9.*	
T1/NL/S8	The *if statement* in the inner *loop* (*line*s 22–25) is used so that the product will be aligned properly.	
T1/NL/S9	If the product is a single digit, it is displayed with an extra space before it.	
4.7	MINIMIZING NUMERICAL ERRORS	
T1/MNE/S1	Numeric errors involving floating-point numbers are inevitable.	
T1/MNE/S2	This section discusses how to minimize such errors through an example. Read this section that discusses how to minimize such errors through an example.	Dec/St/Co
T1/MNE/S3	*Listing 4.5* presents an example that *sum*s a series that starts with *0.01* and ends with *1.0.* See *Listing 4.5* that presents an example that *sum*s a series that starts with *0.01* and ends with *1.0.*	Dec/St/Co
T1/MNE/S4	The numbers in the series will increment by *0.01*, as follows: *0.01 + 0.02 + 0.03* and so on.	
T1/MNE/S5	The output of the program appears in *Figure 4.7.* See the output of the program in *Figure 4.7.* Dec/St/Co	
T1/MNE/S6	The *for loop* (*line*s *9–10*) repeatedly adds the control variable *i* to the *sum*.	
T1/MNE/S7	This variable, which begins with *0.01*, is incremented by *0.01* after each iteration.	
T1/MNE/S8	The *loop* terminates when *i* exceeds *1.0.*	
T1/MNE/S9	The *for loop initial action* can be any *statement*, but it is often used to initialize a control variable.	
T1/MNE/S10	From this example, you can see that a control variable can be a *float* type. From this example, see that a control variable can be a *float* type.	Dec/St/Co
T1/MNE/S11	In fact, it can be any data type.	
T1/MNE/S12	The exact *sum* should be *50.50,* but the answer is *50.499985.*	
T1/MNE/S13	The result is not precise because computers use a fixed number of bits to represent floating-point numbers, and thus cannot represent some *floating*-point numbers exactly.	
T1/MNE/S14	If you change *float* in the program to *double* as follows, you should see a slight improvement in precision because a *double* variable takes sixty-four *bits*, whereas a *float* variable takes thirty-two *bits*. If you change *float* in the program to *double* as follows, notice a slight improvement in precision because a *double* variable takes sixty-four *bits*, whereas a *float* variable takes thirty-two *bits*.	Dec/St/Co
T1/MNE/S15	However, you will be stunned to see that the result is actually *49.50000000000003.* However, see that the result is actually *49.50000000000003.*	Dec/St/Co
T1/MNE/S16	What went wrong? There is a mistake.	Int/Qu/St
T1/MNE/S17	If you *print* out *i* for each iteration in the *loop*, you will see that the last *i* is slightly larger than 1 (not exactly 1).	
T1/MNE/S18	This causes the last *i* not to be added into *sum*.	
T1/MNE/S19	The fundamental problem is that the *floating-point* numbers are represented by approximation.	
T1/MNE/S20	Errors commonly occur.	
T1/MNE/S21	There are two ways to fix the problem.	

T1/MNE/S22	Minimizing errors by processing large numbers first. Minimize errors by processing large numbers first.	Dec/St/Co
T1/MNE/S23	Using an integer *count* to ensure that all the numbers are processed. Use an integer *count* to ensure that all the numbers are processed.	Dec/St/Co
T1/MNE/S24	To minimize errors, add numbers from *1.0, 0.99,* down to *0.1,* as follows: To minimize errors, you can add numbers from *1.0, 0.99,* down to *0.1,* as follows:	Imp/Co/St
T1/MNE/S25	To ensure that all the items are added to *sum*, use an integer variable to *count* the items. To minimize errors, you can use an integer variable to *count* the items.	Imp/Co/St
T1/MNE/S26	Here is the new *loop*. See here for the new *loop.*	Dec/St/Co
T1/MNE/S27	After this *loop*, *sum* is *50.50000000000003.*	
4.8	CASE STUDIES	
T1/CS/S1	Control *statements* are fundamental in programming.	
T1/CS/S2	The ability to write control *statements* is essential in learning *Java* programming.	
T1/CS/S3	If you can write programs using *loops*, you know how to program!	
T1/CS/S4	For this reason, this section presents three additional examples of how to solve problems using *loops*. For this reason, read this section that presents three additional examples of how to solve problems using *loops*.	Dec/St/Co
4.8.1	EXAMPLE: FINDING THE GREATEST COMMON DIVISOR	
T1/FGCD/S1	This section presents a program that prompts the user to enter two positive integers and finds their greatest common divisor. Read this section that presents a program that prompts the user to enter two positive integers and finds their greatest common divisor.	Dec/St/Co
T1/FGCD/S2	The greatest common divisor of two integers *4* and *2* is *2.*	
T1/FGCD/S3	The greatest common divisor of two integers *16* and *24* is *8.*	
T1/FGCD/S4	How do you find the greatest common divisor? Find the greatest common divisor	Dec/St/Co
T1/FGCD/S5	Let the two input integers be *n1* and *n2.* You can let the two input integers be *n1* and *n2.*	Imp/Co/St
T1/FGCD/S6	You know that number 1 is a common divisor, but it may not be the greatest common divisor.	
T1/FGCD/S7	So you can check whether *k (*for *k = 2, 3, 4* and so on) is a common divisor for *n1* and *n2*, until *k* is greater than *n1* or *n2.* Check whether *k (*for *k = 2, 3, 4* and so on) is a common divisor for *n1* and *n2*, until *k* is greater than *n1* or *n2.*	Dec/St/Co
T1/FGCD/S8	Store the common divisor in a variable named *gcd.* You can store the common divisor in a variable named *gcd.*	Imp/Co/St
T1/FGCD/S9	Initially, *gcd* is *1.*	
T1/FGCD/S10	Whenever a new common divisor is found, it becomes the new *gcd.*	
T1/FGCD/S11	When you have checked all the possible common divisors from *2* up to *n1* or *n2*, the value in variable *gcd* is the greatest common divisor.	Dec/St/Co
T1/FGCD/S12	The idea can be translated into the following *loop*:	Dec/St/Co

	Translate the idea into the following *loop*	
T1/FGCD/S13	The complete program is given in *Listing* 4.6, and a sample run of the program is shown in *Figure 4.8.* See the complete program in *Listing 4.6* and a sample run of the program in *Figure 4.8.*	
T1/FGCD/S14	The program finds the greatest common divisor for two integers.	
T1/FGCD/S15	How did you write this program? Describe how you wrote this program.	Dec/St/Co
T1/FGCD/S16	Did you immediately begin to write the code?	
T1/FGCD/S17	No. You didn't immediately begin to write the code.	Int/Qu/St
T1/FGCD/S18	It is important to think before you type. Think before you type.	Dec/St/Co
T1/FGCD/S19	Thinking enables you to generate a logical solution for the problem without concern about how to write the code.	
T1/FGCD/S20	Once you have a logical solution, type the code to translate the solution into a *Java* program. Once you have a logical solution, you can type the code to translate the solution into a *Java* program.	Imp/Co/St
T1/FGCD/S21	The translation is not unique.	
T1/FGCD/S22	For example, you could use a *for loop* to rewrite the code as follows: For example, use a *for loop* to rewrite the code as follows:	Dec/St/Co
T1/FGCD/S23	A problem often has multiple solutions.	
T1/FGCD/S24	The *GCD* problem can be solved in many ways. You can solve The *GCD* problem in many ways.	
T1/FGCD/S25	*Exercise 4.15* suggests another solution. See *Exercise 4.15* that suggests another solution.	
T1/FGCD/S26	A more efficient solution is to use the classic Euclidean algorithm. For a more efficient solution, use the classic Euclidean algorithm.	Dec/St/Co
T1/FGCD/S27	See *http://www.cut-the-knot.org/blue/Euclid.shtml* for more information. You can see *http://www.cut-the-knot.org/blue/Euclid.shtml* for more information.	Imp/Co/St
T1/FGCD/S28	You might think that a divisor for a number *n1* cannot be greater than *n1 / 2.*	
T1/FGCD/S29	So you would attempt to improve the program using the following *loop*:	
T1/FGCD/S30	This revision is wrong.	
T1/FGCD/S31	Can you find the reason? Find the reason.	Int/Qu/Co
T1/FGCD/S32	See *Review Question 4.9* for the answer. You can see *Review Question 4.9* for the answer.	Imp/Co/St
4.8.2	EXAMPLE: FINDING THE SALES AMOUNT	
T1/FSA/S1	You have just started a sales job in a department store.	
T1/FSA/S2	Your pay consists of a base salary and a commission.	
T1/FSA/S3	The base salary is *$5,000.*	
T1/FSA/S4	The scheme shown below is used to determine the commission rate.	

T1/FSA/S5	Your goal is to earn *$30,000* a year.	
T1/FSA/S6	This section writes a program that finds the minimum amount of sales you have to generate in order to make *$30,000.* Read this section that writes a program that finds the minimum amount of sales you have to generate in order to make *$30,000.*	Dec/St/Co
T1/FSA/S7	Since your base salary is *$5,000*, you have to make *$25,000* in commissions to earn *$30,000* a year.	
T1/FSA/S8	What is the sales amount *for* a *$25,000* commission? Find the sales amount *for* a *$25,000* commission.	Dec/St/Co
T1/FSA/S9	If you know the sales amount, the commission can be computed as follows. If you know the sales amount, compute the commission as follows.	Dec/St/Co
T1/FSA/S10	This suggests that you can try to find the *salesAmount* to match a given commission through incremental approximation. Try to find the *salesAmount* to match a given commission through incremental approximation.	Dec/St/Co
T1/FSA/S11	For a *salesAmount* of *$0.01 (1 cent),* find commission.	
T1/FSA/S12	If commission is less than *$25,000*, increment *salesAmount* by *0.01* and find commission again.	
T1/FSA/S13	If commission is still less than *$25,000,* repeat the process until it is greater than or equal to *$25,000.*	
T1/FSA/S14	This is a tedious job for humans, but it is exactly what a computer is good for.	
T1/FSA/S15	You can write a *loop* and let a computer execute it painlessly. Write a *loop* and let a computer execute it painlessly.	Dec/St/Co
T1/FSA/S16	The idea can be translated into the following *loop*: Translate the idea into the following loop:	Dec/St/Co
T1/FSA/S17	The complete program is given in *Listing 4.7*, and a sample run of the program is shown in *Figure 4.9.* Read the complete program given in *Listing 4.7*, and a sample run of the program in *Figure 4.9.*	
T1/FSA/S18	The *do-while loop* (*line*s 12–24) is used to repeatedly compute commission for an incremental *salesAmount.*	
T1/FSA/S19	The *loop* terminates when commission is greater than or equal to a constant *COMMISSION_SOUGHT.*	
T1/FSA/S20	In *Exercise 4.17*, you will rewrite this program to let the user enter *COMMISSION_SOUGHT* dynamically from an input dialog. In *Exercise 4.17*, rewrite this program to let the user enter *COMMISSION_SOUGHT* dynamically from an input dialog.	Dec/St/Co
T1/FSA/S21	You can improve the performance of this program by estimating a higher *INITIAL_SALES_AMOUNT (e.g., 25000).* Improve the performance of this program by estimating a higher *INITIAL_SALES_AMOUNT (e.g., 25000).*	Dec/St/Co
T1/FSA/S22	What is wrong if *salesAmount* is incremented after the commission is computed, as follows? Find the mistake if *salesAmount* is incremented after the commission is computed, as follows?	Dec/St/Co
T1/FSA/S23	The change is erroneous because *salesAmount* is 1 cent more than is needed for the commission when the *loop* ends.	
T1/FSA/S24	This is a common error in *loops*, known as the *off-by-one error*.	
4.8.3	EXAMPLE: DISPLAYING A PYRAMID OF NUMBERS	Dec/St/Co
T1/DPN/S1	This section presents a program that prompts the user to enter an integer from *1* to *15* and displays a pyramid. Read this section that presents a program that prompts the user to enter an integer from *1* to *15* and displays a pyramid.	Dec/St/Co
T1/DPN/S2	If the input integer is *12*, for example, the output is shown in *Figure 4.10.* If the input integer is *12*, for example, see the output in *Figure 4.10.*	
T1/DPN/S3	The program uses *nested loops* to *print* numbers in a triangular pattern.	

T1/DPN/S4	Your program receives the input *for* an integer *(numberOfLines)* that represents the total number of lines.	
T1/DPN/S5	It displays all the lines one by one.	
T1/DPN/S6	Each line has three parts.	
T1/DPN/S7	The first part comprises the spaces before the numbers;	
T1/DPN/S8	The second part, the leading numbers, such as 3 2 1 in *line* 3.	
T1/DPN/S9	And the last part, the ending numbers, such as 2 3 in *line* 3.	
T1/DPN/S10	Each number occupies three spaces.	
T1/DPN/S11	Display an empty space before a double-digit number, and display two empty spaces before a single-digit number.	
T1/DPN/S12	You can use an outer *loop* to control the lines. Use an outer *loop* to control the lines.	Dec/St/Co
T1/DPN/S13	At the n^{th} row, there are (*numberOfLines – n)*3* leading spaces, the leading numbers are *n, n-1, … 1,* and the ending numbers are *2, ..., n.*	
T1/DPN/S14	You can use three separate inner *loops* to *print* each part. Use three separate inner *loops* to *print* each part.	Dec/St/Co
T1/DPN/S15	Here is the algorithm for the problem. Read the algorithm for the problem.	Dec/St/Co
T1/DPN/S16	The complete program is given in *Listing 4.8.* See the complete program in *Listing 4.8.*	Dec/St/Co
T1/DPN/S17	The program uses the *print method (lines 20, 24, and 28)* to display a *string* to the console.	
T1/DPN/S18	The conditional expression *(num >= 10) ? " " + num : " " + num* in *line*s 24 and 28 returns a *string* with a single empty space before the number if the number is greater than or equal to 10, and otherwise returns a *string* with two empty spaces before the number.	
T1/DPN/S19	Printing patterns like this one and the ones in Exercises 4.18 and 4.19 is a good exercise for practicing *loop* control *statements*.	
T1/DPN/S20	The key is to understand the pattern and to describe it using *loop* control variables. Understand the pattern and to describe it using *loop* control variables.	
T1/DPN/S21	The last line in the outer *loop* (*line* 31), *System.out.println(),* does not have any argument in the *method*.	
T1/DPN/S22	This call moves the cursor to the next line.	
4.9	KEYWORDS BREAK AND CONTINUE	
T1/KBC/S1	Two *statements*, *break* and *continue*, can be used in *loop statements* to provide the *loop* with additional control. Use the two *statements*, *break* and *continue* in *loop statements* to provide the *loop* with additional control.	Dec/St/Co
T1/KBC/S2	*break* immediately ends the innermost *loop* that contains it.	
T1/KBC/S3	It is generally used with an *if statement*. Use it with an *if statement.*	Dec/St/Co
T1/KBC/S4	*continue* only ends the current iteration.	
T1/KBC/S5	Program control goes to the end of the *loop* body.	
T1/KBC/S6	This keyword is generally used with an *if statement*.	Dec/St/Co

	Use this keyword with an *if statement.*	
T1/KBC/S7	You have already used the keyword *break* in a *switch statement*.	
T1/KBC/S8	You can also use *break* and *continue* in a *loop*. Use *break* and *continue* in a *loop*.	Dec/St/Co
T1/KBC/S9	*Listings 4.9* and *4.10* present two programs to demonstrate the effect of the *break* and *continue* keywords in a *loop*. See *Listings 4.9* and *4.10* that present two programs which demonstrate the effect of the *break* and *continue* keywords in a *loop*.	
T1/KBC/S10	The program in *Listing 4.9* adds the integers from *1 to 20* in this order to *sum* until *sum* is greater than or equal to *100.*	
T1/KBC/S11	Without the *if statement* (*line* 10), the program calculates the *sum* of the numbers from *1 to 20.*	
T1/KBC/S12	But with the *if statement*, the *loop* terminates when the *sum* becomes greater than or equal to *100.*	
T1/KBC/S13	The output of the program is shown in *Figure 4.11(a).*	
T1/KBC/S14	If you changed the *if statement* as shown below, the output would resemble that in *Figure 4.11(b).*	
T1/KBC/S15	In this case, the *if* condition will never be true.	
T1/KBC/S16	Therefore, the *break statement* will never be executed.	
T1/KBC/S17	The program in *Listing 4.10* adds all the integers from *1 to 20* except *10 and 11* to *sum*.	
T1/KBC/S18	With the *if statement* in the program (*line* 9), the *continue statement* is executed when number becomes *10 or 11*.	
T1/KBC/S19	The *continue statement* ends the current iteration so that the rest of the *statement* in the *loop* body is not executed; there*fo*re, number is not added to *sum* when it is 10 or 11.	
T1/KBC/S20	The output of the program is shown in *Figure 4.12(a).* See the output of the program in *Figure 4.12(a).*	Dec/St/Co
T1/KBC/S21	Without the *if statement* in the program, the output would look like *Figure 4.12(b).* Without the *if statement* in the program, see how the output would look like in *Figure 4.12(b).*	
T1/KBC/S22	In this case, all of the numbers are added to *sum*, even when number is *10 or 11.*	
T1/KBC/S23	Therefore, the result is *210*, which is *21* more than it was with the *if statement*.	

Appendix 3B

Text 2 – Features of Metaphor of Mood

Note:
1. For every table entry with metaphorical clauses with Metaphor of Mood, the metaphorical sentence precedes the congruent sentence.

Keys:
1. Dec/St/Co denotes Declarative clauses realizing Statement and Command
2. Int/Qu/St denotes Interrogative clauses realizing Question and Statement
3. Int/Qu/Co denotes Interrogative clauses realizing Question and Command
4. Imp/Co/St denotes Imperative clauses realizing Command and Statement

CHAPTER 9	FEELING A LITTLE *LOOP*Y	Code
TITLE	A *LOOP* FOR EVERY OCCASION (LEO)	
T2/LEO/S1	Have you ever been talking to someone and it seems like he or she is saying the same thing over and over? There is probably a time when you talk to someone and it seems like he or she is saying the same thing over and over.	Int/Qu/St
T2/LEO/S2	I mean, you keep listening, and they keep talking, and it all sounds the same. I mean, you keep listening, and they keep talking, doesn't it all sound the same?	
T2/LEO/S3	And they talk somemore and you listen somemore and you wonder if it will ever end! And they talk somemore and you listen somemore and you wonder, will it ever end?	
T2/LEO/S4	Congratulations, you just experienced a perfect example of a verbal *loop*! Congratulations, experience a perfect example of a verbal *loop!*	Dec/St/Co
T2/LEO/S5	In *Java*, a *loop* is a programming construct that enables you to repeat a section of code over and over, much like my conversation example.	
T2/LEO/S6	*Loops* are very valuable in *Java* because they enable you to tightly control repetitive functions.	
T2/LEO/S7	Three type of *loops* are used in *Java*: *for loops*, *while loops* and *do loops*.	
TITLE	GETTING REDUNDANT WITH THE *FOR LOOP* (GRL)	
T2/GRL/S1	Let's pretend NASA used *Java* applets to control the launch of the space shuttle. Pretend NASA used *Java* applets to control the launch of the space shuttle.	Dec/St/Co
T2/GRL/S2	Any ideas on how controllers would initiate the launch sequence? Think of how controllers would initiate the launch sequence.	Int/Qu/Co
T2/GRL/S3	With *loops*!	
T2/GRL/S4	Counting down from ten to one is a piece of cake with a *loop*.	

T2/GRL/S5	Granted, without a *loop* it wouldn't be too tough either, but it would require some unnecessary code.	
T2/GRL/S6	Following is code to perform the launch sequence without the use of a *loop*. See the following for the code to perform the launch sequence without the use of a *loop*.	Dec/St/Co
T2/GRL/S7	And now the *loop* version: And now see the *loop* version:	Dec/St/Co
T2/GRL/S8	See what I mean about tightening up the code? This is what I mean about tightening up the code.	Int/Qu/St
T2/GRL/S9	You probably wonder exactly how the *loop* code works. Think of how does the loop code work.	Dec/St/Co
T2/GRL/S10	This code relies on a *for loop*, which is the most structured type of *loop* supported by *Java*.	
T2/GRL/S11	*For loops* repeat a section of code a fixed number of times.	
T2/GRL/S12	Following is the syntax for the *for loop*: See the following for the syntax for the *for loop.*	
T2/GRL/S13	The *for loop* repeats the *Statement* the number of time determined by the *InitializationExpression, LoopCondition* and *StepExpression:*	
T2/GRL/S14	The *InitializationExpression* is used to initialize a *loop* control variable. Use the *InitializationExpression* to initialize a *loop* control variable.	Dec/St/Co
T2/GRL/S15	The *LoopCondition* compares the *loop* control variable to some limit or value.	
T2/GRL/S16	The *StepExpression* specifies how the *loop* control variable should be modified before the next iteration of the *loop*.	
T2/GRL/S17	Let's take a look at the NASA launch sequence code again to make some sense of this stuff. Take a look at the NASA launch sequence code again to make some sense of this stuff.	Dec/St/Co
T2/GRL/S18	In this code the *InitializationExpression* is *int i310*, which is evaluated initially before the *loop* begins.	
T2/GRL/S19	This is the code you use to prime the *loop* and get it ready. Use this code to prime the *loop* and get it ready.	Dec/St/Co
T2/GRL/S20	The *LoopCondition* is *i>0*, which is a *Boolean* test that is performed before each iteration of the *loop*.	
T2/GRL/S21	If the *Boolean* test result is true, the *Statement* is executed, which in this case prints the current value of *i.*	
T2/GRL/S22	After each iteration the *StepExpression* is evaluated, which is *i--*.	
T2/GRL/S23	This serves to decrement *i* after each iteration, and ultimately proves the countdown.	
T2/GRL/S24	The *loop* continues to iterate and print numbers as *i* counts down to 0.	
T2/GRL/S25	After *i* reaches 0, the *LoopCondition* test fails (i>0), so the *loop* bails out without printing any more numbers.	
T2/GRL/S26	Whew, that explanation seemed a little long-winded, and that's coming from the person that wrote it!	
T2/GRL/S27	Unfortunately, it isn't always easy to verbalize the flow of program code.	
T2/GRL/S28	This is why it's easy to fall back on figures.	
T2/GRL/S29	Just ask Ross Perot, who isn't a *Java* programmer but who nonetheless relied on diagrams and illustrations to help us grasp his big plans for the presidency. You can ask Ross Perot, who isn't a *Java* programmer but who nonetheless relied on diagrams and illustrations to help us grasp his big plans for the presidency.	Imp/Co/St
T2/GRL/S30	You can feel safe and secure knowing that I'm not running *for* president or trying to help you visualize my answer to	Dec/St/Co

	global trade. Be safe and secure knowing that I'm not running *for* president or trying to help you visualize my answer to global trade.	
T2/GRL/S31	I just want to help you learn how *loops* work!	
T2/GRL/S32	To help you visualize the *looping* process, take a look at the following figure. To help you visualize the *looping* process, you can take a look at the following figure.	Imp/Co/St
T2/GRL/S33	Notice in the figure that *Statement 1* and *Statement 2* will be repeatedly executed as long as the *loop* condition is true. You can notice in the figure that *Statement 1* and *Statement 2* will be repeatedly executed as long as the *loop* condition is true.	Imp/Co/St
T2/GRL/S34	When the *loop* condition goes false, the program falls out of the *loop* and executes *Statement 3.*	
T2/GRL/S35	The previous figure alludes to the fact that a *loop* can execute multiple *statements*.	
T2/GRL/S36	*Loops* can execute as many *statements* as they want, provided curly braces ({}) enclose the *statements.*	
T2/GRL/S37	If you recall, this grouping of *statements* is known as a *compound statement* and was used in the previous chapter when dealing with *if-else* branches. Recall that this grouping of *statements* is known as a *compound statement* and was used in the previous chapter when dealing with *if-else* branches.	Dec/St/Co
T2/GRL/S38	Following is an example of a *for loop* with a *compound statement:* See the following for an example of a *for loop* with a *compound statement:*	Dec/St/Co
T2/GRL/S39	This code calculates the squares of the numbers 1 through 10, stores them in an array, and prints each one.	
T2/GRL/S40	Notice that the *loop counter (i)* is used as the index *(i-1)* into the squares array. You can notice that the *loop counter (i)* is used as the index *(i-1)* into the squares array.	Imp/Co/St
T2/GRL/S41	This is a very popular way to handle arrays.	
T2/GRL/S42	It is necessary to subtract 1 in this case because all *Java* array indexes start with 0, which means they are zero based. Subtract 1 in this case because all *Java* array indexes start with 0, which means they are zero based.	Dec/St/Co
T2/GRL/S43	It might be worth nothing that although zero-based arrays were used in other programming languages in the 1980s and before, they have nothing to *do* with the 80s movie Less than Zero or the 80s hit song saved by Zero.	
T2/GRL/S44	Rest assured I would be the first to tell you if they did! You can be rest assured I would be the first to tell you if they did!	Imp/Co/St
TITLE	LOOPING FOR JUST A LITTLE WHILE (LJLW)	
T2/LJLW/S1	Like the *for loop*, the *while loop* has a *loop* condition that controls the number of times a *loop* is repeated.	
T2/LJLW/S2	However, the *while loop* has no *initialization* or *step expression.*	
T2/LJLW/S3	A *for loop* is like one of those friends who tells you a story three or four times and then waits for a response, whereas a *while loop* is like one of those friends who continues to repeat himself as long as you continue to listen.	
T2/LJLW/S4	They're both annoying, but in different ways.	
T2/LJLW/S5	Not the *loops*, the people!	
T2/LJLW/S6	Following is the syntax *for* the *while loop*, which should make its usage a little more clear: See the following for the syntax for the *while loop*, which should make its usage a little more clear:	Dec/St/Co
T2/LJLW/S7	If the *Boolean LoopCondition* evaluates to true, the *Statement* is executed.	
T2/LJLW/S8	When the *Statement* finishes executing, the *LoopCondition* is tested again and the process repeats itself.	

T2/LJLW/S9	This continues until the *LoopCondition* evaluates to false, in which case the *loop* immediately bails out.	
T2/LJLW/S10	Because the *while loop* has no *step expression*, it is important to make sure that the *Statement* somehow impacts the *LoopCondition*. Because the *while loop* has no *step expression*, make sure that the *Statement* somehow impacts the *LoopCondition*.	Dec/St/Co
T2/LJLW/S11	Otherwise, it is possible *for* the *loop* to repeat infinitely, which is usually a bad thing.	
T2/LJLW/S12	Following is a simple example of an infinite *while loop*: See the following for a simple example of an infinite *while loop*:	Dec/St/Co
T2/LJLW/S13	Because the *loop* condition in this example is permanently set to true, the *loop* will repeat infinitely, or at least until you manually terminate the programme.	
T2/LJLW/S14	Infinite *loops* are extremely dangerous because they can result in your computer overheating.	
T2/LJLW/S15	Just kidding!	
T2/LJLW/S16	Actually, infinite *loops* are useful in some situations.	
T2/LJLW/S17	They are never truly infinite because you can typically terminate one by shutting down the application or applet containing it.	
T2/LJLW/S18	You can think of the *while loop* as a more general *for loop*. Think of the *while loop* as a more general *for loop*.	Dec/St/Co
T2/LJLW/S19	To understand what I mean by this, check out the following code. To understand what I mean by this, you can check out the following code.	Imp/Co/St
T2/LJLW/S20	This is the NASA launch sequence implemented using a *while loop* instead of a *for loop*.	
T2/LJLW/S21	Because *while loops* don't have *initialization* expressions, the *initialization* of the *counter* variable *i* had to be performed before the *loop*.	
T2/LJLW/S22	Likewise, the *step expression i* had to be performed within the *Statement* part of the *loop*.	
T2/LJLW/S23	Regardless of the structural differences, this *while loop* is functionally equivalent to the *for loop* you saw earlier in the chapter.	
T2/LJLW/S24	If a *for loop* can do everything a *while loop* can and in a more organized way, then why do we need *while loops*? Even when a *for loop* can do everything a *while loop* can and in a more organized way, we still need *while loops.*	Int/Qu/St
T2/LJLW/S25	Because there is a time and a place for everything, and in many situations you have no need *for initialization* and *step expressions.*	
T2/LJLW/S26	A *for loop* is overkill in situations like this.	
T2/LJLW/S27	Even more importantly, a *while loop* is much more readable than a *for loop* when you have no need *for initialization* and *step expressions.*	
T2/LJLW/S28	Consider the following example: You can consider the following example:	Imp/Co/St
T2/LJLW/S29	This code demonstrates how a *while loop* could be used to ask a question and patiently wait for the correct answer. See this code that demonstrates how a *while loop* could be used to ask a question and patiently wait for the correct answer.	Dec/St/Co
T2/LJLW/S30	The *loop* repeats itself as long as the *Boolean* variable correct is false.	
T2/LJLW/S31	This results in the code repeating the question as many times as necessary until the user guesses the correct answer.	

T2/LJLW/S32	The details of the methods *askQuestion()* and *isCorrect()* aren't important for this example.	
T2/LJLW/S33	Just assume that they somehow present the user with a question, retrieve an answer, and then judge the correctness of the answer. You can assume that they somehow present the user with a question, retrieve an answer, and then judge the correctness of the answer.	Imp/Co/St
T2/LJLW/S34	The main concern is that the *isCorrect () method* returns a *Boolean* value that indicates whether or not the answer is correct.	
T2/LJLW/S35	In this example, it is impossible to know how many times the user will miss the answer and need the question repeated.	
T2/LJLW/S36	For this reason, the structured *step expression* of a *for loop* wouldn't be of much use.	
T2/LJLW/S37	While *loops* are perfect in situations where you don't know ahead of time how many times a *loop* needs to be repeated.	
T2/LJLW/S38	If you aren't completely satisfied with *while loops*, however, there is one other option. If you aren't completely satisfied with *while loops*, however, consider one other option.	Dec/St/Co
TITLE	TO *DO* , OR NOT TO *DO* (TDNTD)	
T2/TDNTD/S1	The *while loop* has a very close relative known as the *do loop*, or *do-while loop*, that is surprisingly similar to the *while loop*.	
T2/TDNTD/S2	Because you're becoming pretty *loop* savvy, I'll show you the syntax *for* the *do-while loop* first and see if you can figure out how it works. Because you're becoming pretty *loop* savvy, take note on the syntax *for* the *do-while loop* first and see if you can figure out how it works.	Dec/St/Co
T2/TDNTD/S3	Give up? If you can't figure it out, the answer is as follows: Give up now because the answer is as follows:	 Int/Qu/St Int/Qu/Co
T2/TDNTD/S4	The *do-while loop* is basically a *while loop* with the *LoopCondition* moved to the end.	
T2/TDNTD/S5	Why is this necessary? This is necessary.	Int/Qu/St
T2/TDNTD/S6	Because there are some situations where you would like the *Statement* to execute before evaluating the *LoopCondition*, instead of afterward.	
T2/TDNTD/S7	This also guarantees that the *Statement* is executed at least once, regardless of the *LoopCondition*.	
T2/TDNTD/S8	Let's take a look at the question and answer example implemented using a *do-while loop*: You can take a look at the question and answer example implemented using a *do-while loop*:	Imp/Co/St
T2/TDNTD/S9	The code really isn't much different than before, except that you no longer need to *initialise* the correct variable.	
T2/TDNTD/S10	It is always initially set during the first pass through the *loop*.	
T2/TDNTD/S11	Although both types of *loops* accomplish the goal of this example, the *do-while loop* is a better fit because of its structure more closely mimics the function of the code.	
T2/TDNTD/S12	What do I mean by this? Read on to know what I mean by this. The following describes what I mean by this.	 Int/Qu/Co Int/Qu/St
T2/TDNTD/S13	Well, if you "read" the code, it is saying "ask the question and if the answer is not correct, ask it again."	

T2/TDNTD/S14	This makes more sense than if it read “if the answer is not correct, ask the question and then check the answer again.”	
T2/TDNTD/S15	Admittedly, this is a subtle difference, but a large part of successful programming is keeping things logical and straightforward.	
T2/TDNTD/S16	You won’t always succeed because sometimes code gets complicated regardless of how you construct it, but using *loops* intelligently is a good start.	
TITLE	APPLET COUNTDOWN (AC)	
T2/AC/S1	Have you ever visited a Web page that directed you to another page, but informed you that if you waited a few second sit would automatically take you there? There is probably a time when you visit a Web page that directed you to another page, but informed you that if you waited a few second sit would automatically take you there.	Int/Qu/St
T2/AC/S2	I used to run across these pages and wonder how you could make a page wait a few seconds and then automatically navigate to a new page.	
T2/AC/S3	After I started programming in *Java*, I realised what a trivial task this is.	
T2/AC/S4	In this section you use your knowledge of *loops* to build a “countdown” applet that counts down from ten to one and then navigates to a new Web page. In this section, use your knowledge of *loops* to build a “countdown” applet that counts down from ten to one and then navigates to a new Web page.	Dec/St/Co
T2/AC/S5	The following figure shows the *Countdown* applet in action. See the following figure that shows the *Countdown* applet in action.	Dec/St/Co
T2/AC/S6	When the applet finishes counting down, it navigates to the web page identified by the page applet parameter.	
T2/AC/S7	As an example, what web site could be better than NASA’s to demonstrate how this applet works? As an example, think of what web site that could be better than NASA’s to demonstrate how this applet works.	Dec/St/Co
T2/AC/S8	Following is NASA’s Web site, to which the *Countdown* applet will take you after it finishes its countdown. See the following for NASA’s Web site to which the *Countdown* applet will take you after it finishes its countdown.	Dec/St/Co
T2/AC/S9	To understand how the *Countdown* applet works, let’s first take a look at the *Countdown. Html* Web page that contains the embedded applet. To understand how the *Countdown* applet works, you can first take a look at the *Countdown. Html* Web page that contains the embedded applet.	Imp/Co/St
T2/AC/S10	All this stuff should look pretty familiar to you by now.	
T2/AC/S11	The main thing on which I want you to focus is the page parameter, which is defined as: Focus on the page parameter, which is defined as:	Dec/St/Co
T2/AC/S12	Notice that the value of the page parameter is set to http://www.nasa.gov, which is the URL of NASA’s Web site. You can notice that the value of the page parameter is set to http://www.nasa.gov, which is the URL of NASA’s Web site.	Imp/Co/St
T2/AC/S13	Changing this value enables you to change the page that is loaded after the applet finishes counting down.	
T2/AC/S14	This page could have easily been set as a variable within the applet code, but a recompile would be required to change the page.	
T2/AC/S15	That is the beauty of applet parameters.	
T2/AC/S16	They enable you to customize the function of applets without doing any real programming!	Dec/St/Co

	Customize the function of applets without doing any real programming!	
T2/AC/S17	Let's move on to the actual code required of the countdown applet. You can move on to the actual code required of the countdown applet.	Imp/Co/St
T2/AC/S18	Unfortunately, the *Countdown* applet requires some code that is a little beyond the lesson, so I don't expect all of this applet to make sense to you.	
T2/AC/S19	However, you can download the complete source code for the applet from the book's companion Web site, which was mentioned a little earlier in this section.	
T2/AC/S20	Also, the core mechanics of the applet are very straightforward and should be familiar to you from your recent study of *loops*.	
T2/AC/S21	Following is the *run() method* in the *Countdown* applet class, which forms the heart of the applet: See the following for the *run() method* in the *Countdown* applet class, which forms the heart of the applet:	Dec/St/Co
T2/AC/S22	Ouch, that looks a little messy!	
T2/AC/S23	Try not to get intimidated by any code that doesn't look familiar. You must try not to get intimidated by any code that doesn't look familiar.	Imp/Co/St
T2/AC/S24	Just concentrate on the *loop* code. You can just concentrate on the *loop* code.	Imp/Co/St
T2/AC/S25	As you can see, the *for loop* counts down from 10 to 1 just like the *Countdown* code you saw earlier in the chapter. See that the *for loop* counts down from 10 to 1 just like the *Countdown* code you saw earlier in the chapter.	Dec/St/Co
T2/AC/S26	The *Statement* part of this *for loop* is completely new territory, however.	
T2/AC/S27	The call to the *repaint () method* is necessary to update the applet's window with the new *countdown* number.	
T2/AC/S28	The call to the *thread.sleep() method* results in the applet waiting one second, which effectively pauses the *countdown* for one second between numbers.	
T2/AC/S29	When the *for loop* finishes, the code gets the page applet parameter and proceed to navigate to the Web page identified by it.	
T2/AC/S30	The code required to navigate to the Web page is probably pretty strange looking to you because it has to deal with *exceptions.*	
T2/AC/S31	*Exceptions* are errors caused by unforeseen problems such as your computer running out of memory, your modem coming unplugged, spilling coffee on your keyboard, hurling your monitor out the window, and so on.	
T2/AC/S32	I'll explain *exceptions* as you encounter them throughout the book. Be ready for explanation of *exceptions* as you encounter them throughout the book.	Dec/St/Co
T2/AC/S33	The complete source code for the *countdow*n applet is as follows. See the complete source code for the *countdow*n applet as follows.	Dec/St/Co
T2/AC/S34	Although this is a longer program than you are accustomed to seeing, a lot of it should look familiar to you.	
T2/AC/S35	For example, the *paint() method* code is very similar to the code used in the *DateTime* applet from Chapter 4, "constructing Applets of Your Own."	
T2/AC/S36	On the other hand, the *start() method* is entirely new and is related to the applet's use of *threads.*	
T2/AC/S37	You don't need to understand it fully at this point.	
TITLE	BREAKING AWAY (BA)	

T2/BA/S1	If you recall from the previous chapter, each case section of a *switch* branch ends with a *break statement*. Recall from the previous chapter, each case section of a *switch* branch ends with a *break statement*.	Dec/St/Co
T2/BA/S2	Following is an example to recap: See the following for an example to recap:	Dec/St/Co
T2/BA/S3	The purpose of the *break statement* in this example is to bail out of the *switch* branch so that no other code is executed.	
T2/BA/S4	The *break statement* serves a similar purpose in *loops*.	
T2/BA/S5	It breaks out of a *loop* regardless of the *loop* condition.	
T2/BA/S6	Following is an example of circumventing an infinite *loop* with a *break statement*: See the following for an example of circumventing an infinite *loop* with a *break statement*:	Dec/St/Co
T2/BA/S7	Without the assistance of the *break statement*, this *while loop* would continue forever thanks to the permanent *true loop* condition.	
T2/BA/S8	The *break statement* sidesteps this problem by breaking out of the *loop* after one hundred iterations (0-99).	
T2/BA/S9	Of course, it is rare that you would purposely create an infinite *loop* and then use a *break statement* to bail out of it.	
T2/BA/S10	However, the *break statement* can be very useful in some tricky *loops* when you need to exit at an otherwise inconvenient time.	
T2/BA/S11	A close relative of the *break statement* is the *continue statement*, which is used to skip to the next iteration of a *loop*.	
T2/BA/S13	The following example shows how a *continue statement* can be used to *print* only the even numbers between 1 and 100: See the following example that shows how a *continue statement* can be used to *print* only the even numbers between 1 and 100:	Dec/St/Co
T2/BA/S14	Having trouble seeing how this one works? You might have trouble seeing how this one works.	Int/Qu/St
T2/BA/S15	Think back to the *modulus operator* (%), which returns the remainder of a division. You can think back to the *modulus operator* (%), which returns the remainder of a division.	Imp/Co/St
T2/BA/S16	Now consider what the remainder of a division by 2 yields for even and odd numbers. Now, you can consider what the remainder of a division by 2 yields for even and odd numbers.	Imp/Co/St
T2/BA/S17	Aha!	
T2/BA/S18	Even numbers divided by 2 always yield a remainder of 0, and odd numbers always leave a remainder of 1!	
T2/BA/S19	The example code exploits this characteristic of even and odd numbers to skip to the next iteration bypasses the *println()* call, which prevents odd numbers from being printed. See that the example code exploits this characteristic of even and odd numbers to skip to the next iteration bypasses the *println()* call, which prevents odd numbers from being printed.	Dec/St/Co
T2/BA/S20	Pretty tricky!	
TITLE	THE LEAST YOU NEED TO KNOW (LNK)	
T2/LNK/S1	Computers are often called upon to perform tasks we humans find to be utterly redundant.	
T2/LNK/S2	As dull as some humans can be, I guarantee you computers are much duller when it comes to repeating the same thing over and over.	
T2/LNK/S3	*Java* enables you to build programs that repeat themselves through the use of *loops*.	

T2/LNK/S4	The different types of *loops* basically perform the same function.	
T2/LNK/S5	They repeat a section of code over and over.	
T2/LNK/S6	Let's go over the main points you learned about *loops* in this chapter. You can go over the main points you learned about *loops* in this chapter.	Imp/Co/St
T2/LNK/S7	*Loops* can execute as many *statements* as you want them to, provided the *statements* are grouped together as a single *compound statement* enclosed by curly braces ({}).	
T2/LNK/S8	A *for loop* is used to repeat a section of code a given number of iterations. Use a *for loop* to repeat a section of code a given number of iterations.	Dec/St/Co
T2/LNK/S9	A *while loop* is a more general *for loop*.	
T2/LNK/S10	A *do while* is a *while loop* with the *loop* condition moved to the end.	
T2/LNK/S11	The *break statement* is used to break out of a *loop* regardless of the *loop* condition. Use the *break statement to break out of a loop* regardless of the *loop* condition.	Dec/St/Co
T2/LNK/S12	The *continue statement* is used to skip to the next iteration of a *loop*. Use the *continue statement* to skip to the next iteration of a *loop*.	Dec/St/Co

Text 1 – Semantic Expansion of Declarative Clauses realizing Statement and Command

Note:
1. Semantic Expansion of Declarative Clauses in text is coded as “Dec/St/Co” which denotes Declarative clauses realizing Statement and Command.

Key:
1. R : Realization
2. M : Metaphorical
3. C : Congruent

Eg.	Label	R	Declarative Mood Sentences
1	T1/INT/S4	M	Using a *loop statement*, you simply tell the computer to *print* a *string* a hundred times without having to code the *print statement* a hundred times.
		C	Using a *loop statement*, tell the computer to *print* a *string* a hundred times without having to code the *print statement* a hundred times.
2	T1/LCD/S2	M	If you want the user to decide whether to take another question, you can use a confirmation dialog to control the *loop*.
		C	If you want the user to decide whether to take another question, use a confirmation dialog to control the *loop*.
3	T1/LCD/S3	M	A confirmation dialog can be created using the following *statement*.
		C	Create a confirmation dialog using the following *statement.*
4	T1/LCD/S7	M	You can rewrite *Listing 4.1* using a confirmation dialog to let the user decide whether to continue the next question.
		C	Rewrite *Listing 4.1* using a confirmation dialog to let the user decide whether to continue the next question.
5	T1/LSV/S9	M	A sample run of the program is shown in *Figure 4.3*.
		C	See *Figure 4.3* for a sample run of the program
6	T1/DWL/S2	M	Its syntax is given below:
		C	See below for its syntax:
7	T1/DWL/S3	M	Its execution flow chart is shown in *Figure 4.4.*
		C	See *Figure 4.4* for its execution flow chart.
8	T1/DWL/S11	M	For example, you can rewrite the *while loop* in *Listing 4.2* using a *do-while loop*, as shown in *Listing 4.3*.
		C	For example, rewrite the *while loop* in *Listing 4.2* using a *do-while loop*, as shown in *Listing 4.3.*
9	T1/FL/S2	M	A *for loop* can be used to simplify the preceding *loop*.
		C	Use a *for loop* to simplify the preceding *loop*.
10	T1/FL/S3	M	In general, the syntax of a *for loop* is as shown below.
		C	In general, see below for the syntax of a *for loop.*
11	T1/FL/S5	M	The *for loop statement* starts with the keyword *for*, followed by a pair of parentheses enclosing *initial-action*, *loop-continuation-condition*, and *action-after-each-iteration*, and followed by the *loop* body enclosed inside braces.
		C	Start the *for loop statement* with the keyword *for,* enclose *initial-action, loop-continuation-condition* with a pair of parenthesis, and enclose the *loop* body inside braces.
12	T1/FL/S6	M	*Initial-action*, *loop-continuation-condition*, and *action-after-each-iteration* are separated by semicolons.
		C	Separate *Initial-action*, *loop-continuation-condition*, and *action-after-each-iteration* with semicolons.
13	T1/FL/S7	M	A *for loop* generally uses a variable to control how many times the *loop* body is executed and when the *loop* terminates.
		C	Use a variable in a *for loop* to control how many times the *loop* body is executed and when the *loop* terminates.
14	T1/FL/S8	M	This variable is referred to as a control variable.

		C	Refer this variable as a control variable.
15	T1/FL/S10	M C	For example, the following *for loop* prints "Welcome to *Java*!" a hundred times. See the following to understand how *for loop* prints "Welcome to *Java*!" a hundred times.
16	T1/FL/S11	M C	The flow chart of the *statement* is shown in *Figure 4.5(b)*. See *Figure 4.5(b)* for the flow chart of the *statement.*
17	T1/FL/S23	M C	If there is only one *statement* in the *loop* body, as in this example, the braces can be omitted. If there is only one *statement* in the *loop* body, as in this example, omit the braces.
18	T1/FL/S24	M C	The control variable must always be declared inside the control structure of the *loop* or before the *loop*. Declare the control variable inside the control structure of the *loop* or before the *loop*.
19	T1/FL/S25	M C	The *loop* control variable is used only in the *loop*, and not elsewhere. Use the *loop* control variable only in the *loop*, and not elsewhere.
20	T1/FL/S28	M C	For example, you cannot reference *i* outside the *for loop* in the preceding code, because it is declared inside the *for loop*. Do not reference *i* outside the *for loop* in the preceding code, because it is declared inside the *for loop*.
21	T1/WLU/S1	M C	The *while loop* and *for loop* are called pre-test *loops* because the continuation condition is checked be*fo*re the *loop* body is executed. Refer the *while loop* and *for loop as* pre-test *loops* because the continuation condition is checked be*fo*re the *loop* body is executed.
22	T1/WLU/S2	M C	The *do-while loop* is called a post-test *loop* because the condition is checked after the *loop* body is executed. Refer The *do-while loop as* a post-test *loop* because the condition is checked after the *loop* body is executed
23	T1/WLU/S4	M C	That is, you can write a *loop* in any of these three forms. Write a *loop* in any of these three forms.
24	T1/WLU/S5	M C	For example, a *while loop* in (a) in the following figure can always be converted into the *for loop* in (b): See the following figure on how , a *while loop* in (a) can always be converted into the *for loop* in (b):
25	T1/WLU/S6	M C	A *for loop* in (a) in the next figure can generally be converted into the *while loop* in (b) except in certain special cases (see Review Question 4.12 *for* such a case): See the next figure on how A *for loop* in (a) can generally be converted into the *while loop* in (b) except in certain special cases (see Review Question 4.12 *for* such a case):
26	T1/WLU/S8	M C	In general, a *for loop* may be used if the number of repetitions is known, as, for example, when you need to *print* a message a hundred times. In general, use a *for loop* used if the number of repetitions is known, as, for example, when you need to *print* a message a hundred times.
27	T1/WLU/S9	M C	A *while loop* may be used if the number of repetitions is not known, as in the case of reading the numbers until the input is 0. Use a *while loop* if the number of repetitions is not known, as in the case of reading the numbers until the input is 0.
28	T1/WLU/S10	M C	A *do-while loop* can be used to replace a *while loop* if the *loop* body has to be executed before the continuation condition is tested. Use a *do-while loop* to replace a *while loop* if the *loop* body has to be executed before the continuation condition is tested.
29	T1/NL/S3	M C	*Listing 4.4* presents a program that uses *nested for loops* to *print* a multiplication table, as shown in *Figure 4.6*. See Listing 4.4 that presents a program that uses *nested for loops* to *print* a multiplication table, as shown in *Figure 4.6*
30	T1/MNE/S2	M C	This section discusses how to minimize such errors through an example. Read this section that discusses how to minimize such errors through an example.
31	T1/MNE/S3	M C	*Listing 4.5* presents an example that *sum*s a series that starts with *0.01* and ends with *1.0.* See *Listing 4.5* that presents an example that *sum*s a series that starts with *0.01* and ends with *1.0.*
32	T1/MNE/S10	M C	From this example, you can see that a control variable can be a *float* type. From this example, see that a control variable can be a *float* type.
33	T1/MNE/S14	M C	If you change *float* in the program to *double* as follows, you should see a slight improvement in precision because a *double* variable takes sixty-four *bits*, whereas a *float* variable takes thirty-two *bits*. If you change *float* in the program to *double* as follows, notice a slight improvement in precision because a *double* variable takes sixty-four *bits*, whereas a *float* variable takes thirty-two *bits*.
34	T1/MNE/S15	M C	However, you will be stunned to see that the result is actually *49.50000000000003.* However, see that the result is actually *49.50000000000003.*
35	T1/MNE/S22	M C	Minimizing errors by processing large numbers first. Minimize errors by processing large numbers first.

36	T1/MNE/S23	M C	Using an integer *count* to ensure that all the numbers are processed. Use an integer *count* to ensure that all the numbers are processed.
37	T1/MNE/S26	M C	Here is the new *loop*. See here for the new *loop.*
38	T1/CS/S4	M C	For this reason, this section presents three additional examples of how to solve problems using *loops*. For this reason, read this section that presents three additional examples of how to solve problems using *loops*.
39	T1/FGCD/S1	M C	This section presents a program that prompts the user to enter two positive integers and finds their greatest common divisor. Read this section that presents a program that prompts the user to enter two positive integers and finds their greatest common divisor.
40	T1/FGCD/S4	M C	How do you find the greatest common divisor? Find the greatest common divisor
41	T1/FGCD/S7	M C	So you can check whether *k (*for *k = 2, 3, 4* and so on) is a common divisor for *n1* and *n2*, until *k* is greater than *n1* or *n2.* Check whether *k (*for *k = 2, 3, 4* and so on) is a common divisor for *n1* and *n2*, until *k* is greater than *n1* or *n2.*
42	T1/FGCD/S12	M C	The idea can be translated into the following *loop*: Translate the idea into the following *loop*
43	T1/FGCD/S15	M C	How did you write this program? Describe how you wrote this program.
44	T1/FGCD/S18	M C	It is important to think before you type. Think before you type.
45	T1/FGCD/S22	M C	For example, you could use a *for loop* to rewrite the code as follows: For example, use a *for loop* to rewrite the code as follows:
46	T1/FGCD/S26	M C	A more efficient solution is to use the classic Euclidean algorithm. For a more efficient solution, use the classic Euclidean algorithm.
47	T1/FSA/S6	M C	This section writes a program that finds the minimum amount of sales you have to generate in order to make *$30,000.* Read this section that writes a program that finds the minimum amount of sales you have to generate in order to make *$30,000.*
48	T1/FSA/S8	M C	What is the sales amount *for* a *$25,000* commission? Find the sales amount *for* a *$25,000* commission.
49	T1/FSA/S9	M C	If you know the sales amount, the commission can be computed as follows. If you know the sales amount, compute the commission as follows.
50	T1/FSA/S10	M C	This suggests that you can try to find the *salesAmount* to match a given commission through incremental approximation. Try to find the *salesAmount* to match a given commission through incremental approximation.
51	T1/FSA/S15	M C	You can write a *loop* and let a computer execute it painlessly. Write a *loop* and let a computer execute it painlessly.
52	T1/FSA/S16	M C	The idea can be translated into the following *loop*: Translate the idea into the following loop:
53	T1/FSA/S20	M C	In *Exercise 4.17*, you will rewrite this program to let the user enter *COMMISSION_SOUGHT* dynamically from an input dialog. In *Exercise 4.17*, rewrite this program to let the user enter *COMMISSION_SOUGHT* dynamically from an input dialog.
54	T1/FSA/S21	M C	You can improve the performance of this program by estimating a higher *INITIAL_SALES_AMOUNT (e.g., 25000).* Improve the performance of this program by estimating a higher *INITIAL_SALES_AMOUNT (e.g., 25000).*
55	T1/FSA/S22	M C	What is wrong if *salesAmount* is incremented after the commission is computed, as follows? Find the mistake if *salesAmount* is incremented after the commission is computed, as follows?
56	T1/DPN/S1	M C	This section presents a program that prompts the user to enter an integer from *1* to *15* and displays a pyramid. Read this section that presents a program that prompts the user to enter an integer from *1* to *15* and displays a pyramid.
57	T1/DPN/S12	M C	You can use an outer *loop* to control the lines. Use an outer *loop* to control the lines.
58	T1/DPN/S14	M	You can use three separate inner *loops* to *print* each part.

		C	Use three separate inner *loops* to *print* each part.
59	T1/DPN/S15	M C	Here is the algorithm for the problem. Read the algorithm for the problem.
60	T1/DPN/S16	M C	The complete program is given in *Listing 4.8*. See the complete program in *Listing 4.8.*
61	T1/KBC/S1	M C	Two *statements*, *break* and *continue*, can be used in *loop statements* to provide the *loop* with additional control. Use the two *statements*, *break* and *continue* in *loop statements* to provide the *loop* with additional control.
62	T1/KBC/S3	M C	It is generally used with an *if statement*. Use it with an *if statement.*
63	T1/KBC/S6	M C	This keyword is generally used with an *if statement*. Use this keyword with an *if statement.*
64	T1/KBC/S8	M C	You can also use *break* and *continue* in a *loop*. Use *break* and *continue* in a *loop*.
65	T1/KBC/S20	M C	The output of the program is shown in *Figure 4.12(a)*. See the output of the program in *Figure 4.12(a)*.

Appendix 4B

Text 2 – Semantic Expansion of Declarative Clauses realizing Statement and Command

Note:
1. Semantic Expansion of Declarative Clauses in text is coded as "Dec/St/Co" which denotes Declarative clauses realizing Statement and Command.

Key:
1. R : Realization
2. M : Metaphorical
3. C : Congruent

Eg.	Label	R	Declarative Mood Sentences
1	T2/LEO/S4	M C	Congratulations, you just experienced a perfect example of a verbal *loop*! Congratulations, experience a perfect example of a verbal *loop!*
2	T2/GRL/S1	M C	Let's pretend NASA used *Java* applets to control the launch of the space shuttle. Pretend NASA used *Java* applets to control the launch of the space shuttle.
3	T2/GRL/S6	M C	Following is code to perform the launch sequence without the use of a *loop*. See the following for the code to perform the launch sequence without the use of a *loop*.
4	T2/GRL/S7	M C	And now the *loop* version: And now see the *loop* version:
5	T2/GRL/S9	M C	You probably wonder exactly how the *loop* code works. Think of how does the loop code work.
6	T2/GRL/S14	M C	The *InitializationExpression* is used to initialize a *loop* control variable. Use the *InitializationExpression* to initialize a *loop* control variable.
7	T2/GRL/S17	M C	Let's take a look at the NASA launch sequence code again to make some sense of this stuff. Take a look at the NASA launch sequence code again to make some sense of this stuff.
8	T2/GRL/S19	M C	This is the code you use to prime the *loop* and get it ready. Use this code to prime the *loop* and get it ready.
9	T2/GRL/S30	M C	You can feel safe and secure knowing that I'm not running *for* president or trying to help you visualize my answer to global trade. Be safe and secure knowing that I'm not running *for* president or trying to help you visualize my answer to global trade.
10	T2/GRL/S37	M	If you recall, this grouping of *statements* is known as a *compound statement* and was used in the previous chapter when dealing with *if-else* branches.

		C	Recall that this grouping of *statements* is known as a *compound statement* and was used in the previous chapter when dealing with *if-else* branches.
11	T2/GRL/S38	M C	Following is an example of a *for loop* with a *compound statement:* See the following for an example of a *for loop* with a *compound statement:*
12	T2/GRL/S42	M C	It is necessary to subtract 1 in this case because all *Java* array indexes start with 0, which means they are zero based. Subtract 1 in this case because all *Java* array indexes start with 0, which means they are zero based.
13	T2/LJLW/S6	M C	Following is the syntax *for* the *while loop*, which should make its usage a little more clear: See the following for the syntax for the *while loop*, which should make its usage a little more clear:
14	T2/LJLW/S10	M C	Because the *while loop* has no *step expression*, it is important to make sure that the *Statement* somehow impacts the *LoopCondition*. Because the *while loop* has no *step expression*, make sure that the *Statement* somehow impacts the *LoopCondition*.
15	T2/LJLW/S12	M C	Following is a simple example of an infinite *while loop*: See the following for a simple example of an infinite *while loop*:
16	T2/LJLW/S18	M C	You can think of the *while loop* as a more general *for loop*. Think of the *while loop* as a more general *for loop*.
17	T2/LJLW/S29	M C	This code demonstrates how a *while loop* could be used to ask a question and patiently wait for the correct answer. See this code that demonstrates how a *while loop* could be used to ask a question and patiently wait for the correct answer.
18	T2/LJLW/S38	M C	If you aren't completely satisfied with *while loops*, however, there is one other option. If you aren't completely satisfied with *while loops*, however, consider one other option.
19	T2/TDNTD/S2	M C	Because you're becoming pretty *loop* savvy, I'll show you the syntax *for* the *do-while loop* first and see if you can figure out how it works. Because you're becoming pretty *loop* savvy, take note on the syntax *for* the *do-while loop* first and see if you can figure out how it works.
20	T2/AC/S4	M C	In this section you use your knowledge of *loops* to build a "countdown" applet that counts down from ten to one and then navigates to a new Web page. In this section, use your knowledge of *loops* to build a "countdown" applet that counts down from ten to one and then navigates to a new Web page.
21	T2/AC/S5	M C	The following figure shows the *Countdown* applet in action. See the following figure that shows the *Countdown* applet in action.
22	T2/AC/S7	M C	As an example, what web site could be better than NASA's to demonstrate how this applet works? As an example, think of what web site that could be better than NASA's to demonstrate how this applet works.
23	T2/AC/S8	M C	Following is NASA's Web site, to which the *Countdown* applet will take you after it finishes its countdown. See the following for NASA's Web site to which the *Countdown* applet will take you after it finishes its countdown.
24	T2/AC/S11	M C	The main thing on which I want you to focus is the page parameter, which is defined as: Focus on the page parameter, which is defined as:
25	T2/AC/S16	M C	They enable you to customize the function of applets without doing any real programming! Customize the function of applets without doing any real programming!

26	T2/AC/S21	M C	Following is the *run() method* in the *Countdown* applet class, which forms the heart of the applet: See the following for the *run() method* in the *Countdown* applet class, which forms the heart of the applet:
27	T2/AC/S25	M C	As you can see, the *for loop* counts down from 10 to 1 just like the *Countdown* code you saw earlier in the chapter. See that the *for loop* counts down from 10 to 1 just like the *Countdown* code you saw earlier in the chapter.
28	T2/AC/S32	M C	I'll explain *exceptions* as you encounter them throughout the book. Be ready for explanation of *exceptions* as you encounter them throughout the book.
29	T2/AC/S33	M C	The complete source code for the *countdow*n applet is as follows. See the complete source code for the *countdow*n applet as follows.
30	T2/BA/S1	M C	If you recall from the previous chapter, each case section of a *switch* branch ends with a *break statement*. Recall from the previous chapter, each case section of a *switch* branch ends with a *break statement*.
31	T2/BA/S2	M C	Following is an example to recap: See the following for an example to recap:
32	T2/BA/S6	M C	Following is an example of circumventing an infinite *loop* with a *break statement*: See the following for an example of circumventing an infinite *loop* with a *break statement*:
33	T2/BA/S13	M C	The following example shows how a *continue statement* can be used to *print* only the even numbers between 1 and 100: See the following example that shows how a *continue statement* can be used to *print* only the even numbers between 1 and 100:
34	T2/BA/S19	M C	The example code exploits this characteristic of even and odd numbers to skip to the next iteration bypasses the *println()* call, which prevents odd numbers from being printed. See that the example code exploits this characteristic of even and odd numbers to skip to the next iteration bypasses the *println()* call, which prevents odd numbers from being printed.
35	T2/LNK/S8	M C	A *for loop* is used to repeat a section of code a given number of iterations. Use a *for loop* to repeat a section of code a given number of iterations.
36	T2/LNK/S11	M C	The *break statement* is used to break out of a *loop* regardless of the *loop* condition. Use the *break statement to break out of a loop* regardless of the *loop* condition.
37	T2/LNK/S12	M C	The *continue statement* is used to skip to the next iteration of a *loop*. Use the *continue statement* to skip to the next iteration of a *loop*.

Appendix 5A

Text 1 – Semantic Expansion of Interrogative Clauses realizing Question and Statement

Note:
1. Semantic Expansion of Interrogative Clauses in text is coded as “Int/Qu/St” which denotes Interrogative clauses realizing Question and Statement.

Key:
1. R : Realization
2. M : Metaphorical
3. C : Congruent

Eg.	Label	R	Sentence
1	T1/LSV/S5	M	Do you need to declare a new variable for each input value?
	T1/LSV/S6	M C	No. There is no need to declare a new variable for each input.
2	T1/MNE/S16	M C	What went wrong? There is a mistake.
3	T1/FGCD/S16	M	Did you immediately begin to write the code?
	T1/FGCD/S17	M C	No. You didn’t immediately begin to write the code.

Appendix 5B

Text 2 – Semantic Expansion of Interrogative Clauses realizing Question and Statement

Note:
1. Semantic Expansion of Interrogative Clauses in text is coded as "Int/Qu/St" which denotes Interrogative clauses realizing Question and Statement.

Key:
1. R : Realization
2. M : Metaphorical
3. C : Congruent

Eg.	Label	R	Sentence
1	T2/LEO/S1	M C	Have you ever been talking to someone and it seems like he or she is saying the same thing over and over? There is probably a time when you talk to someone and it seems like he or she is saying the same thing over and over.
3	T2/GRL/S8	M C	See what I mean about tightening up the code? This is what I mean about tightening up the code.
4	T2/LJLW/S24	M C	If a *for loop* can do everything a *while loop* can and in a more organized way, then why do we need *while loops*? Even when a *for loop* can do everything a *while loop* can and in a more organized way, we still need *while loops.*
5	T2/TDNTD/S3	M C	Give up? If you can't figure it out, the answer is as follows:
6	T2/TDNTD/S5	M C	Why is this necessary? This is necessary.
7	T2/TDNTD/S12	M C	What do I mean by this? The following describes what I mean by this.
8	T2/AC/S1	M C	Have you ever visited a Web page that directed you to another page, but informed you that if you waited a few second sit would automatically take you there? There is probably a time when you visit a Web page that directed you to another page, but informed you that if you waited a few second sit would automatically take you there.
9	T2/BA/S14	M C	Having trouble seeing how this one works? You might have trouble seeing how this one works.

Appendix 5C

Text 1 – Semantic Expansion of Interrogative Clauses realizing Question and Command

Note:
1. Semantic Expansion of Interrogative Clauses in text is coded as "Int/Qu/Co" which denotes Interrogative clauses realizing Question and Command

Key:
1. R : Realization
2. M : Metaphorical
3. C : Congruent

Eg.	Label	R	Sentence
1	T1/FGCD/S31	M	Can you find the reason?
		C	Find the reason.

Appendix 5D

Text 2 – Semantic Expansion of Interrogative Clauses realizing Question and Command

Note:
1. Semantic Expansion of Interrogative Clauses in text is coded as "Int/Qu/Co" which denotes Interrogative clauses realizing Question and Command

Key:
1. R : Realization
2. M : Metaphorical
3. C : Congruent

Eg.	Label	R	Sentence
1	T2/GRL/S2	M C	Any ideas on how controllers would initiate the launch sequence? Think of how controllers would initiate the launch sequence.

Appendix 6A

Text 1 – Semantic Expansion of Imperative Clauses realizing Command and Statement

Note:
1. Semantic Expansion of Imperative Clauses in text is coded as "Imp/Co/St" which denotes Imperative clauses realizing Command and Statement.

Key:
1. R : Realization
2. M : Metaphorical
3. C : Congruent

Eg.	Label	R	Sentence
1	T1/LSV/S7	M	Just use one variable named *data* (*line* 9) to store the input value and use a variable named *sum* (*line* 12) to store the total.
		C	One variable named *data* (*line* 9) is used to store the input value and a variable named *sum* (*line* 12) is used to store the total
2	T1/LSV/S13	M	Note that if the first input read is 0, the *loop* body never executes, and the resulting *sum* is 0.
		C	If the first input read is 0, the *loop* body never executes, and the resulting *sum* is 0.
3	T1/WLU/S7	M	Use the *loop statement* that is most intuitive and comfortable for you.
		C	You can use the *loop statement* that is most intuitive and comfortable for you.
4	T1/MNE/S24	M	To minimize errors, add numbers from *1.0, 0.99,* down to *0.1,* as follows:
		C	To minimize errors, you can add numbers from *1.0, 0.99,* down to *0.1,* as follows:
5	T1/MNE/S25	M	To ensure that all the items are added to *sum*, use an integer variable to *count* the items.
		C	To minimize errors, you can use an integer variable to *count* the items.
6	T1/FGCD/S5	M	Let the two input integers be *n1* and *n2.*
		C	You can let the two input integers be *n1* and *n2.*
7	T1/FGCD/S8	M	Store the common divisor in a variable named *gcd.*
		C	You can store the common divisor in a variable named *gcd.*
8	T1/FGCD/S20	M	Once you have a logical solution, type the code to translate the solution into a *Java* program.
		C	Once you have a logical solution, you can type the code to translate the solution into a *Java* program.
9	T1/FGCD/S27	M	See *http://www.cut-the-knot.org/blue/Euclid.shtml* for more information.
		C	You can see *http://www.cut-the-knot.org/blue/Euclid.shtml* for more information.
10	T1/FGCD/S32	M	See *Review Question 4.9* for the answer.
		C	You can see *Review Question 4.9* for the answer.

Appendix 6B

Text 2 – Semantic Expansion of Imperative Clauses realizing Command and Statement

Note:
1. Semantic Expansion of Imperative Clauses in text is coded as "Imp/Co/St" which denotes Imperative clauses realizing Command and Statement.

Key:
1. R : Realization
2. M : Metaphorical
3. C : Congruent

Eg.	Label	R	Imperative Mood Sentences
1	T2/GRL/S29	M	Just ask Ross Perot, who isn't a *Java* programmer but who nonetheless relied on diagrams and illustrations to help us grasp his big plans for the presidency.
		C	You can ask Ross Perot, who isn't a *Java* programmer but who nonetheless relied on diagrams and illustrations to help us grasp his big plans for the presidency.
2	T2/GRL/S32	M	To help you visualize the *looping* process, take a look at the following figure.
		C	To help you visualize the *looping* process, you can take a look at the following figure.
3	T2/GRL/S33	M	Notice in the figure that *Statement 1* and *Statement 2* will be repeatedly executed as long as the *loop* condition is true.
		C	You can notice in the figure that *Statement 1* and *Statement 2* will be repeatedly executed as long as the *loop* condition is true.
4	T2/GRL/S40	M	Notice that the *loop counter (i)* is used as the index *(i-1)* into the squares array.
		C	You can notice that the *loop counter (i)* is used as the index *(i-1)* into the squares array.
5	T2/GRL/S44	M	Rest assured I would be the first to tell you if they did!
		C	You can be rest assured I would be the first to tell you if they did!
6	T2/LJLW/S19	M	To understand what I mean by this, check out the following code.
		C	To understand what I mean by this, you can check out the following code.
7	T2/LJLW/S28	M	Consider the following example:
		C	You can consider the following example:
8	T2/LJLW/S33	M	Just assume that they somehow present the user with a question, retrieve an answer, and then judge the correctness of the answer.
		C	You can assume that they somehow present the user with a question, retrieve an answer, and then judge the correctness of the answer.
9	T2/TDNTD/S8	M	Let's take a look at the question and answer example implemented using a *do-while loop*:

		C	You can take a look at the question and answer example implemented using a *do-while loop*:
10	T2/AC/S9	M	To understand how the *Countdown* applet works, let's first take a look at the *Countdown. Html* Web page that contains the embedded applet.
		C	To understand how the *Countdown* applet works, you can first take a look at the *Countdown. Html* Web page that contains the embedded applet.
11	T2/AC/S12	M	Notice that the value of the page parameter is set to http://www.nasa.gov, which is the URL of NASA's Web site.
		C	You can notice that the value of the page parameter is set to http://www.nasa.gov, which is the URL of NASA's Web site.
12	T2/AC/S17	M	Let's move on to the actual code required of the countdown applet.
		C	You can move on to the actual code required of the countdown applet.
13	T2/AC/S23	M	Try not to get intimidated by any code that doesn't look familiar.
		C	You must try not to get intimidated by any code that doesn't look familiar.
14	T2/AC/S24	M	Just concentrate on the *loop* code.
		C	You can just concentrate on the *loop* code.
15	T2/BA/S15	M	Think back to the *modulus operator* (%), which returns the remainder of a division.
		C	You can think back to the *modulus operator* (%), which returns the remainder of a division.
16	T2/BA/S16	M	Now consider what the remainder of a division by 2 yields for even and odd numbers.
		C	Now, you can consider what the remainder of a division by 2 yields for even and odd numbers.
17	T2/LNK/S6	M	Let's go over the main points you learned about *loops* in this chapter.
		C	You can go over the main points you learned about *loops* in this chapter.

Appendix 7A

Text 1 – Features of Metaphor of Modality

Keys:
1. Men/Prob denotes Mental Projection clause with Probability
2. Men/Ob denotes Mental Projection clause with Obligation
3. Rel/Prob denotes Relational Projection clause with Probability
4. Rel/Ob denotes Relational Projection clause with Obligation

Note:
1. Projection clauses are in bold and separated with oblique lines (||).

4.1	INTRODUCTION	Code
T1/INT/S1	Suppose that you need to *print* a *string* (e.g., "Welcome to *Java*!") a hundred times.	
T1/INT/S2	**It would be tedious** || to have to write the following *statement* a hundred times.	Rel/Prob
T1/INT/S3	*Java* provides a powerful control structure called a *loop* that controls how many times an operation or a sequence of operations is performed in succession.	
T1/INT/S4	Using a *loop statement*, you simply tell the computer to *print* a *string* a hundred times without having to code the *print statement* a hundred times.	
T1/INT/S5	*Loops* are structures that control repeated executions of a block of *statements*.	
T1/INT/S6	The concept of *looping* is fundamental to programming.	
T1/INT/S7	*Java* provides three types of *loop statements*: *while loops*, *do-while loops*, and *for loops*.	
4.2	THE *WHILE LOOP*	
T1/WL/S1	The syntax *for* the *while loop* is as follows.	
T1/WL/S2	The *while loop* flow chart is shown in *Figure 4.1(a).*	
T1/WL/S3	The part of the *loop* that contains the *statements* to be repeated is called the *loop body*.	
T1/WL/S4	A one-time execution of a *loop* body is referred to as an iteration of the *loop*.	
T1/WL/S5	Each *loop* contains a *loop-continuation* condition, a Boolean expression that controls the execution of the body.	
T1/WL/S6	**It is always evaluated** before the *loop* body is executed.	Rel/Ob
T1/WL/S7	If its evaluation is true, the *loop* body is executed.	
T1/WL/S8	If its evaluation is false, the entire *loop* terminates and the program control turns to the *statement* that follows the *while loop*.	
T1/WL/S9	For example, the following *while loop* prints “Welcome to *Java*!” a hundred times.	
T1/WL/S10	The flow chart of the preceding *statement* is shown in *Figure 4.1(b).*	
T1/WL/S11	The variable *count* is initially 0.	

T1/WL/S12	The *loop* checks whether (*count* < *100*) is true.	
T1/WL/S13	If so, it executes the *loop* body to *print* the message “Welcome to *Java*!” and increments *count* by 1.	
T1/WL/S14	It repeatedly executes the *loop* body until (*count* < *100*) becomes false.	
T1/WL/S15	When (*count* < *100*) is false (i.e., when *count* reaches *100*), the *loop* terminates and the next *statement* after the *loop statement* is executed.	
4.2.1	AN ADVANCED MATH LEARNING TOOL	
T1/AMLT/S1	The Math subtraction learning tool program in *Listing* 3.5, *SubtractionTutor.Java*, generates just one question for each run.	
T1/AMLT/S2	You can use a *loop* to generate questions repeatedly.	
T1/AMLT/S3	*Listing* 4.1 gives a program that generates ten questions and reports the number of correct answers after a student answers all ten questions.	
T1/AMLT/S4	The program also displays the time spent on the test and lists all the questions, as shown in *Figure 4.2*.	
T1/AMLT/S5	The program uses the control variable *count* to control the execution of the *loop*.	
T1/AMLT/S6	*Count* is initially 0 (*line* 6) and is increased by 1 in each iteration (*line* 39).	
T1/AMLT/S7	A subtraction question is displayed and processed in each iteration.	
T1/AMLT/S8	The program obtains the time before the test starts in *line* 7 and the time after the test ends in *line* 45, and computes the test time in *line* 46.	
T1/AMLT/S9	The test time is in milliseconds and is converted to seconds in *line 50*.	
4.2.2	CONTROLLING A *LOOP* WITH A CONFIRMATION DIALOG	
T1/LCD/S1	The preceding example executes the *loop* ten times.	
T1/LCD/S2	If you want the user to decide whether to take another question, you can use a confirmation dialog to control the *loop*.	
T1/LCD/S3	A confirmation dialog can be created using the following *statement*:	
T1/LCD/S4	When a button is clicked, the *method* returns an option value.	
T1/LCD/S5	The value is *JoptionPane.YES_OPTION (0)* for the Yes button, *JoptionPane.NO_OPTION (1)* for the No button, and *JoptionPane.CANCEL_OPTION (2)* for the Cancel button.	
T1/LCD/S6	For example, the following *loop* continues to execute until the user clicks the No or Cancel button.	
T1/LCD/S7	You can rewrite *Listing 4.1* using a confirmation dialog to let the user decide whether to continue the next question.	
4.2.3	CONTROLLING A *LOOP* WITH A SENTINEL VALUE	
T1/LSV/S1	Another common technique for controlling a *loop* is to designate a special value when reading and processing a set of values.	
T1/LSV/S2	This special input value, known as a *sentinel value*, signifies the end of the *loop*.	
T1/LSV/S3	*Listing 4.2* writes a program that reads and calculates the *sum* of an unspecified number of integers.	
T1/LSV/S4	The input 0 signifies the end of the input.	
T1/LSV/S5	Do you need to declare a new variable for each input value?	
T1/LSV/S6	No.	
T1/LSV/S7	Just use one variable named data (*line* 9) to store the input value and use a variable named *sum* (*line* 12) to store the total.	
T1/LSV/S8	Whenever a value is read, assign it to data and added to *sum* (*line* 14) if it is not zero.	

T1/LSV/S9	A sample run of the program is shown in *Figure 4.3.*	
T1/LSV/S10	If data is not 0, it is added to the *sum* (*line* 14) and the next items of input data are read (*line*s 12–19).	
T1/LSV/S11	If data is 0, the *loop* body is no longer executed and the *while loop* terminates.	
T1/LSV/S12	The input value 0 is the sentinel value for this *loop*.	
T1/LSV/S13	Note that if the first input read is 0, the *loop* body never executes, and the resulting *sum* is 0.	
T1/LSV/S14	The program uses a *while loop* to add an unspecified number of integers.	
4.3	THE *DO-WHILE LOOP*	
T1/DWL/S1	The *do-while loop* is a variation of the *while loop*.	
T1/DWL/S2	Its syntax is given below:	
T1/DWL/S3	Its execution flow chart is shown in *Figure 4.4.*	
T1/DWL/S4	The *loop* body is executed first. Then the *loop-continuation-condition* is evaluated.	
T1/DWL/S5	If the evaluation is true, the *loop* body is executed again.	
T1/DWL/S6	If it is false, the *do-while loop* terminates.	
T1/DWL/S7	The major difference between a *while loop* and a *do-while loop* is the order in which the *loop-continuation-condition* is evaluated and the *loop* body executed.	
T1/DWL/S8	The *while loop* and the *do-while loop* have equal expressive power.	
T1/DWL/S9	Sometimes one is a more convenient choice than the other.	
T1/DWL/S10	For example, you can rewrite the *while loop* in *Listing 4.2* using a *do-while loop*, as shown in *Listing 4.3.*	
4.4	THE *FOR LOOP*	
T1/FL/S1	Often you write a *loop* in the following common form.	
T1/FL/S2	A *for loop* can be used to simplify the preceding *loop*.	
T1/FL/S3	In general, the syntax of a *for loop* is as shown below.	
T1/FL/S4	The flow chart of the *for loop* is shown in *Figure 4.5(a)*.	
T1/FL/S5	The *for loop statement* starts with the keyword *for*, followed by a pair of parentheses enclosing *initial-action*, *loop-continuation-condition*, and *action-after-each-iteration*, and followed by the *loop* body enclosed inside braces.	
T1/FL/S6	*Initial-action*, *loop-continuation-condition*, and *action-after-each-iteration* are separated by semicolons.	
T1/FL/S7	A *for loop* generally uses a variable to control how many times the *loop* body is executed and when the *loop* terminates.	
T1/FL/S8	This variable is referred to as a control variable.	
T1/FL/S9	The *initial-action* often initializes a control variable, the *action-after-each-iteration* usually increments or decrements the control variable, and the *loop-continuation-condition* tests whether the control variable has reached a termination value.	
T1/FL/S10	For example, the following *for loop* prints “Welcome to *Java*!” a hundred times.	
T1/FL/S11	The flow chart of the *statement* is shown in *Figure 4.5(b)*.	
T1/FL/S12	The *for loop* initializes *I* to 0, then repeatedly executes the *println statement* and evaluates *i++ while I* is less than *100.*	
T1/FL/S13	The *initial-action, i=0,* initializes the control variable, *i.*	
T1/FL/S14	The *loop-continuation-condition*, *i< 100* is a Boolean expression.	
T1/FL/S15	The expression is evaluated at the beginning of each iteration.	
T1/FL/S16	If this condition is true, execute the *loop* body.	

T1/FL/S17	If it is false, the *loop* terminates and the program control turns to the *line* following the *loop*.	
T1/FL/S18	The *action-after-each-iteration, i++,* is a *statement* that adjusts the control variable.	
T1/FL/S19	This *statement* is executed after each iteration.	
T1/FL/S20	It increments the control variable.	
T1/FL/S21	Eventually, the value of the control variable should force the *loop-continuation-condition* to become false.	
T1/FL/S22	Otherwise the *loop* is infinite.	
T1/FL/S23	If there is only one *statement* in the *loop* body, as in this example, the braces can be omitted.	
T1/FL/S24	The control variable must always be declared inside the control structure of the *loop* or before the *loop*.	
T1/FL/S25	If the *loop* control variable is used only in the *loop*, and not elsewhere.	
T1/FL/S26	**It is good programming practice** || to declare it in the *initial-action* of the *for loop*.	Rel/Prob
T1/FL/S27	If the variable is declared inside the *loop* control structure, it cannot be referenced outside the *loop*.	
T1/FL/S28	For example, you cannot reference *I* outside the *for loop* in the preceding code, because it is declared inside the *for loop*.	
4.5.	WHICH *LOOP* TO USE?	
T1/WLU/S1	The *while loop* and *for loop* are called pre-test *loops* because the continuation condition is checked be*fo*re the *loop* body is executed.	
T1/WLU/S2	The *do-while loop* is called a post-test *loop* because the condition is checked after the *loop* body is executed.	
T1/WLU/S3	The three *for*ms of *loop statements*, *while*, *do-while*, and *for*, are expressively equivalent.	
T1/WLU/S4	That is, you can write a *loop* in any of these three forms.	
T1/WLU/S5	For example, a *while loop* in (a) in the following figure can always be converted into the *for loop* in (b):	
T1/WLU/S6	A *for loop* in (a) in the next *Figure* can generally be converted into the *while loop* in (b) except in certain special cases (see *Review Question 4.12 for* such a case):	
T1/WLU/S7	Use the *loop statement* that is most intuitive and comfortable for you.	
T1/WLU/S8	In general, a *for loop* may be used if the number of repetitions is known, as, for example, when you need to *print* a message a hundred times.	
T1/WLU/S9	A *while loop* may be used if the number of repetitions is not known, as in the case of reading the numbers until the input is 0.	
T1/WLU/S10	A *do-while loop* can be used to replace a *while loop* if the *loop* body has to be executed before the continuation condition is tested.	
4.6.	*NESTED LOOPS*	
T1/NL/S1	*Nested loops* consist of an outer *loop* and one or more inner *loops*.	
T1/NL/S2	Each time the outer *loop* is repeated, the inner *loops* are reentered, and started anew.	
T1/NL/S3	*Listing 4.4* presents a program that uses *nested for loops* to *print* a multiplication table, as shown in *Figure* 4.6.	
T1/NL/S4	The program displays a title (*line* 7) on the first *line* and dashes (-) (*line* 8) on the second *line*.	
T1/NL/S5	The first *for loop* (*line*s 12–13) displays the numbers 1 through 9 on the third *line*.	
T1/NL/S6	The next *loop* (*line*s 18–28) is a *nested for loop* with the control variable *I* in the outer *loop* and *j* in the inner *loop*.	
T1/NL/S7	For each I, the product *I * j* is displayed on a *line* in the inner *loop*, with *j* being *1, 2, 3, …, 9.*	
T1/NL/S8	The *if statement* in the inner *loop* (*line*s 22–25) is used so that the product will be aligned properly.	

T1/NL/S9	If the product is a single digit, it is displayed with an extra space before it.	
4.7	MINIMIZING NUMERICAL ERRORS	
T1/MNE/S1	Numeric errors involving floating-point numbers are inevitable.	
T1/MNE/S2	This section discusses how to minimize such errors through an example.	
T1/MNE/S3	*Listing 4.5* presents an example that *sum*s a series that starts with *0.01* and ends with *1.0.*	
T1/MNE/S4	The numbers in the series will increment by 0.01, as follows: *0.01 + 0.02 + 0.03* and so on.	
T1/MNE/S5	The output of the program appears in *Figure 4.7.*	
T1/MNE/S6	The *for loop* (*line*s 9–10) repeatedly adds the control variable *I* to the *sum*.	
T1/MNE/S7	This variable, which begins with *0.01*, is incremented by *0.01* after each iteration.	
T1/MNE/S8	The *loop* terminates when *I* exceeds *1.0.*	
T1/MNE/S9	The *for loop initial action* can be any *statement*, but it is often used to initialize a control variable.	
T1/MNE/S10	From this example, **you can see that** || a control variable can be a *float* type.	Men/Prob
T1/MNE/S11	In fact, it can be any data type.	
T1/MNE/S12	The exact *sum* should be *50.50,* but the answer is *50.499985.*	
T1/MNE/S13	The result is not precise because computers use a fixed number of bits to represent floating-point numbers, and thus cannot represent some *floating*-point numbers exactly.	
T1/MNE/S14	If you change *float* in the program to *double* as follows, you should see a slight improvement in precision because a *double* variable takes sixty-four *bits*, whereas a *float* variable takes thirty-two *bits*.	
T1/MNE/S15	However, **you will be stunned to see that** || the result is actually *49.50000000000003.*	Men/Prob
T1/MNE/S16	What went wrong?	
T1/MNE/S17	If you *print* out *i* for each iteration in the *loop*, you will see that the last *i* is slightly larger than 1 (not exactly 1).	
T1/MNE/S18	This causes the last *i* not to be added into *sum*.	
T1/MNE/S19	**The fundamental problem is that** || the *floating-point* numbers are represented by approximation.	Rel/Prob
T1/MNE/S20	Errors commonly occur.	
T1/MNE/S21	There are two ways to fix the problem.	
T1/MNE/S22	Minimizing errors by processing large numbers first.	
T1/MNE/S23	Using an integer *count* to ensure that all the numbers are processed.	
T1/MNE/S24	To minimize errors, add numbers from *1.0, 0.99,* down to *0.1,* as follows:	
T1/MNE/S25	To ensure that all the items are added to *sum*, use an integer variable to *count* the items.	
T1/MNE/S26	Here is the new *loop*.	
T1/MNE/S27	After this *loop*, *sum* is *50.50000000000003.*	
4.8	CASE STUDIES	
T1/CS/S1	Control *statements* are fundamental in programming.	
T1/CS/S2	The ability to write control *statements* is essential in learning *Java* programming.	
T1/CS/S3	If you can write programs using *loops*, **you know** || how to program!	Men/Prob
T1/CS/S4	For this reason, this section presents three additional examples of how to solve problems using *loops*.	
4.8.1	EXAMPLE: FINDING THE GREATEST COMMON DIVISOR	

T1/FGCD/S1	This section presents a program that prompts the user to enter two positive integers and finds their greatest common divisor.	
T1/FGCD/S2	The greatest common divisor of two integers *4* and *2* is *2.*	
T1/FGCD/S3	The greatest common divisor of two integers *16* and *24* is *8.*	
T1/FGCD/S4	How do you find the greatest common divisor?	
T1/FGCD/S5	Let the two input integers be *n1* and *n2.*	
T1/FGCD/S6	**You know that** || number 1 is a common divisor, but it may not be the greatest common divisor.	Men/Prob
T1/FGCD/S7	So you can check whether *k (*for *k = 2, 3, 4* and so on) is a common divisor for *n1* and *n2*, until *k* is greater than *n1* or *n2.*	
T1/FGCD/S8	Store the common divisor in a variable named *gcd.*	
T1/FGCD/S9	Initially, *gcd* is *1.*	
T1/FGCD/S10	Whenever a new common divisor is found, it becomes the new *gcd.*	
T1/FGCD/S11	When you have checked all the possible common divisors from *2* up to *n1* or *n2*, the value in variable *gcd* is the greatest common divisor.	
T1/FGCD/S12	The idea can be translated into the following *loop*:	
T1/FGCD/S13	The complete program is given in *Listing* 4.6, and a sample run of the program is shown in *Figure 4.8.*	
T1/FGCD/S14	The program finds the greatest common divisor for two integers.	
T1/FGCD/S15	How did you write this program?	
T1/FGCD/S16	Did you immediately begin to write the code?	
T1/FGCD/S17	No.	
T1/FGCD/S18	**It is important** || to think before you type.	Rel/Prob
T1/FGCD/S19	Thinking enables you to generate a logical solution for the problem without concern about how to write the code.	
T1/FGCD/S20	Once you have a logical solution, type the code to translate the solution into a *Java* program.	
T1/FGCD/S21	The translation is not unique.	
T1/FGCD/S22	For example, you could use a *for loop* to rewrite the code as follows:	
T1/FGCD/S23	A problem often has multiple solutions.	
T1/FGCD/S24	The *GCD* problem can be solved in many ways.	
T1/FGCD/S25	*Exercise 4.15* suggests another solution.	
T1/FGCD/S26	A more efficient solution is to use the classic Euclidean algorithm.	
T1/FGCD/S27	See *http://www.cut-the-knot.org/blue/Euclid.shtml* for more in*for*mation.	
T1/FGCD/S28	**You might think that** || a divisor for a number *n1* cannot be greater than *n1 / 2.*	Men/Prob
T1/FGCD/S29	So you would attempt to improve the program using the following *loop*:	
T1/FGCD/S30	This revision is wrong.	
T1/FGCD/S31	Can you find the reason?	
T1/FGCD/S32	See *Review Question 4.9* for the answer.	
4.8.2	EXAMPLE: FINDING THE SALES AMOUNT	
T1/FSA/S1	You have just started a sales job in a department store.	
T1/FSA/S2	Your pay consists of a base salary and a commission.	

T1/FSA/S3	The base salary is *$5,000.*	
T1/FSA/S4	The scheme shown below is used to determine the commission rate.	
T1/FSA/S5	Your goal is to earn *$30,000* a year.	
T1/FSA/S6	This section writes a program that finds the minimum amount of sales you have to generate in order to make *$30,000.*	
T1/FSA/S7	Since your base salary is *$5,000*, you have to make *$25,000* in commissions to earn *$30,000* a year.	
T1/FSA/S8	What is the sales amount *for* a *$25,000* commission?	
T1/FSA/S9	If you know the sales amount, the commission can be computed as follows:	
T1/FSA/S10	**This suggests that** || you can try to find the *salesAmount* to match a given commission through incremental approximation.	Rel/Prob
T1/FSA/S11	For a *salesAmount* of *$0.01 (1 cent),* find commission.	
T1/FSA/S12	If commission is less than *$25,000*, increment *salesAmount* by *0.01* and find commission again.	
T1/FSA/S13	If commission is still less than *$25,000,* repeat the process until it is greater than or equal to *$25,000.*	
T1/FSA/S14	This is a tedious job for humans, but **it is exactly what** || a computer is good for.	Rel/Prob
T1/FSA/S15	You can write a *loop* and let a computer execute it painlessly.	
T1/FSA/S16	The idea can be translated into the following *loop*:	
T1/FSA/S17	The complete program is given in *Listing* 4.7, and a sample run of the program is shown in *Figure* 4.9.	
T1/FSA/S18	The *do-while loop* (*line*s 12–24) is used to repeatedly compute commission for an incremental *salesAmount.*	
T1/FSA/S19	The *loop* terminates when commission is greater than or equal to a constant *COMMISSION_SOUGHT.*	
T1/FSA/S20	In *Exercise 4.17*, you will rewrite this program to let the user enter *COMMISSION_SOUGHT* dynamically from an input dialog.	
T1/FSA/S21	You can improve the performance of this program by estimating a higher *INITIAL_SALES_AMOUNT (e.g., 25000).*	
T1/FSA/S22	What is wrong if *salesAmount* is incremented after the commission is computed, as follows?	
T1/FSA/S23	The change is erroneous because *salesAmount* is 1 cent more than is needed for the commission when the *loop* ends.	
T1/FSA/S24	This is a common error in *loops*, known as the *off-by-one error*.	
4.8.3	EXAMPLE: DISPLAYING A PYRAMID OF NUMBERS	
T1/DPN/S1	This section presents a program that prompts the user to enter an integer from *1* to *15* and displays a pyramid.	
T1/DPN/S2	If the input integer is *12*, for example, the output is shown in *Figure 4.10.*	
T1/DPN/S3	The program uses *nested loops* to *print* numbers in a triangular pattern.	
T1/DPN/S4	Your program receives the input *for* an integer *(numberOfLines)* that represents the total number of lines.	
T1/DPN/S5	It displays all the lines one by one.	
T1/DPN/S6	Each line has three parts.	
T1/DPN/S7	The first part comprises the spaces before the numbers;	
T1/DPN/S8	The second part, the leading numbers, such as 3 2 1 in *line* 3.	
T1/DPN/S9	And the last part, the ending numbers, such as 2 3 in *line* 3.	
T1/DPN/S10	Each number occupies three spaces.	
T1/DPN/S11	Display an empty space before a double-digit number, and display two empty spaces before a single-digit number.	
T1/DPN/S12	You can use an outer *loop* to control the lines.	

T1/DPN/S13	At the n^{th} row, there are (*numberOfLines – n)*3* leading spaces, the leading numbers are *n, n-1, … 1,* and the ending numbers are *2, …, n.*	
T1/DPN/S14	You can use three separate inner *loops* to *print* each part.	
T1/DPN/S15	Here is the algorithm for the problem	
T1/DPN/S16	The complete program is given in *Listing 4.8.*	
T1/DPN/S17	The program uses the *print method (lines 20, 24, and 28)* to display a *string* to the console.	
T1/DPN/S18	The conditional expression *(num >= 10) ? " " + num : " " + num* in *line*s 24 and 28 returns a *string* with a single empty space before the number if the number is greater than or equal to 10, and otherwise returns a *string* with two empty spaces before the number.	
T1/DPN/S19	Printing patterns like this one and the ones in Exercises 4.18 and 4.19 is a good exercise for practicing *loop* control *statements*.	
T1/DPN/S20	The key is to understand the pattern and to describe it using *loop* control variables.	
T1/DPN/S21	The last line in the outer *loop* (*line* 31), *System.out.println(),* does not have any argument in the *method*.	
T1/DPN/S22	This call moves the cursor to the next line.	
4.9	KEYWORDS BREAK AND CONTINUE	
T1/KBC/S1	Two *statements*, *break* and *continue*, can be used in *loop statements* to provide the *loop* with additional control.	
T1/KBC/S2	*break* immediately ends the innermost *loop* that contains it.	
T1/KBC/S3	**It is generally used** || with an *if statement*.	Rel/Ob
T1/KBC/S4	*continue* only ends the current iteration.	
T1/KBC/S5	Program control goes to the end of the *loop* body.	
T1/KBC/S6	**This keyword is generally used** || with an *if statement*.	Rel/Ob
T1/KBC/S7	You have already used the keyword *break* in a *switch statement*.	
T1/KBC/S8	You can also use *break* and *continue* in a *loop*.	
T1/KBC/S9	*Listings 4.9* and *4.10* present two programs to demonstrate the effect of the *break* and *continue* keywords in a *loop*.	
T1/KBC/S10	The program in *Listing 4.9* adds the integers from *1 to 20* in this order to *sum* until *sum* is greater than or equal to *100.*	
T1/KBC/S11	Without the *if statement* (*line* 10), the program calculates the *sum* of the numbers from *1 to 20.*	
T1/KBC/S12	But with the *if statement*, the *loop* terminates when the *sum* becomes greater than or equal to *100.*	
T1/KBC/S13	The output of the program is shown in *Figure 4.11(a).*	
T1/KBC/S14	If you changed the *if statement* as shown below, the output would resemble that in *Figure 4.11(b).*	
T1/KBC/S15	In this case, the *if* condition will never be true.	
T1/KBC/S16	Therefore, the *break statement* will never be executed.	
T1/KBC/S17	The program in *Listing 4.10* adds all the integers from *1 to 20* except *10 and 11* to *sum*.	
T1/KBC/S18	With the *if statement* in the program (*line* 9), the *continue statement* is executed when number becomes *10 or 11.*	
T1/KBC/S19	The *continue statement* ends the current iteration so that the rest of the *statement* in the *loop* body is not executed; there*fo*re, number is not added to *sum* when it is 10 or 11.	
T1/KBC/S20	The output of the program is shown in *Figure 4.12(a).*	
T1/KBC/S21	Without the if *statement* in the program, the output would look like *Figure 4.12(b).*	

T1/KBC/S22	In this case, all of the numbers are added to *sum*, even when number is *10 or 11.*	
T1/KBC/S23	Therefore, the result is *210*, which is *21* more than it was with the *if statement*.	

Appendix 7B

Text 2 – Features of Metaphor of Modality

Keys:
1. Men/Prob denotes Mental Projection clause with Probability
2. Men/Ob denotes Mental Projection clause with Obligation
3. Rel/Prob denotes Relational Projection clause with Probability
4. Rel/Ob denotes Relational Projection clause with Obligation

Note:
1. Projection clauses are in bold and separated with oblique lines (||).

Chapter 9	Feeling a Little Loopy	Code
Title	A loop for Every Occasion (LEO)	
T2/LEO/S1	Have you ever been talking to someone and **it seems like** || he or she is saying the same thing over and over?	Rel/Prob
T2/LEO/S2	**I mean** ||, you keep listening, and they keep talking, and it all sounds the same.	Men/Ob
T2/LEO/S3	And they talk somemore and you listen somemore and **you wonder** || if it will ever end!	Men/Prob
T2/LEO/S4	Congratulations, **you just experienced** || a perfect example of a verbal loop!	Men/Prob
T2/LEO/S5	In Java, a loop is a programming construct that enables you to repeat a section of code over and over, much like my conversation example.	
T2/LEO/S6	Loops are very valuable in Java because they enable you to tightly control repetitive functions.	
T2/LEO/S7	Three type of loops are used in Java: for loops, while loops and do loops.	
Title	Getting redundant with the for loop (GRL)	
T2/GRL/S1	Let's pretend NASA used Java applets to control the launch of the space shuttle.	
T2/GRL/S2	Any ideas on how controllers would initiate the launch sequence?	
T2/GRL/S3	With loops!	
T2/GRL/S4	Counting down from ten to one is a piece of cake with a loop.	
T2/GRL/S5	Granted, without a loop it wouldn't be too tough either, but it would require some unnecessary code.	
T2/GRL/S6	Following is code to perform the launch sequence without the use of a loop.	
T2/GRL/S7	And now the loop version:	
T2/GRL/S8	See what I mean about tightening up the code?	
T2/GRL/S9	**You probably wonder** || exactly how the loop code works.	Men/Prob

T2/GRL/S10	This code relies on a for loop, which is the most structured type of loop supported by Java.	
T2/GRL/S11	For loops repeat a section of code a fixed number of times.	
T2/GRL/S12	Following is the syntax for the for loop:	
T2/GRL/S13	The for loop repeats the Statement the number of time determined by the initializationexpression, LoopCondition and StepExpression:	
T2/GRL/S14	The InitializationExpression is used to initialize a loop control variable.	
T2/GRL/S15	The Loopcondition compares the loop control variable to some limit or value.	
T2/GRL/S16	The StepExpression specifies how the loop control variable should be modified before the next iteration of the loop.	
T2/GRL/S17	Let's take a look at the NASA launch sequence code again to make some sense of this stuff:	
T2/GRL/S18	In this code the Initializationexpression is int i310, which is evaluated initially before the loop begins.	
T2/GRL/S19	This is the code you use to prime the loop and get it ready.	
T2/GRL/S20	The LoopCondition is i>0, which is a Boolean test that is performed before each iteration of the loop.	
T2/GRL/S21	If the Boolean test result is true, the Statement is executed, which in this case prints the current value of i.	
T2/GRL/S22	After each iteration the StepExpression is evaluated, which is i--.	
T2/GRL/S23	This serves to decrement I after each iteration, and ultimately proves the countdown.	
T2/GRL/S24	The loop continues to iterate and print numbers as i counts down to 0.	
T2/GRL/S25	After i reaches 0, the LoopCondition test fails (i>0), so the loop bails out without printing any more numbers.	
T2/GRL/S26	Whew, **that explanation seemed** || a little long-winded, and that's coming from the person that wrote it!	Men/Prob
T2/GRL/S27	Unfortunately, **it isn't always easy** || to verbalize the flow of program code.	Rel/Prob
T2/GRL/S28	**This is why it's easy** || to fall back on figures.	Rel/Prob
T2/GRL/S29	Just ask Ross Perot, who isn't a Java programmer but who nonetheless relied on diagrams and illustrations to help us grasp his big plans for the presidency.	
T2/GRL/S30	**You can feel safe and secure knowing that** || I'm not running for president or trying to help you visualize my answer to global trade.	Men/Prob
T2/GRL/S31	**I just want** || to help you learn how loops work!	Men/Ob
T2/GRL/S32	To help you visualize the looping process, take a look at the following figure.	
T2/GRL/S33	Notice in the figure that Statement 1 and Statement 2 will be repeatedly executed as long as the loop condition is true.	
T2/GRL/S34	When the loop condition goes false, the program falls out of the loop and executes statement 3.	
T2/GRL/S35	The previous figure alludes to the fact that a loop can execute multiple statements.	
T2/GRL/S36	Loops can execute as many statements as they want, provided curly braces ({}) enclose the statements.	
T2/GRL/S37	If you recall, this grouping of statements is known as a compound statement and was used in the previous chapter when dealing with if-else branches.	
T2/GRL/S38	Following is an example of a for loop with a compound statement:	
T2/GRL/S39	This code calculates the squares of the numbers 1 through 10, stores them in an array, and prints each one.	
T2/GRL/S40	Notice that the loop counter (i) is used as the index (i-1) into the squares array.	
T2/GRL/S41	**This is a very popular way** || to handle arrays.	Rel/Prob
T2/GRL/S42	**It is necessary ||** to subtract 1 in this case because all Java array indexes start with 0, which means they are zero based.	Rel/Prob

T2/GRL/S43	**It might be worth nothing that** || although zero-based arrays were used in other programming languages in the 1980s and before, they have nothing to do with the 80s movie Less than Zero or the 80s hit song saved by Zero.	Rel/Ob
T2/GRL/S44	Rest assured I would be the first to tell you if they did!	
Title	Looping for Just a Little While (LJLW)	
T2/LJLW/S1	Like the for loop, the while loop has a loop condition that controls the number of times a loop is repeated.	
T2/LJLW/S2	However, the while loop has no initialization or step expression.	
T2/LJLW/S3	A for loop is like one of those friends who tells you a story three or four times and then waits for a response, whereas a while loop is like one of those friends who continues to repeat himself as long as you continue to listen.	
T2/LJLW/S4	They're both annoying, but in different ways.	
T2/LJLW/S5	Not the loops, the people!	
T2/LJLW/S6	Following is the syntax for the while loop, which should make its usage a little more clear:	
T2/LJLW/S7	If the Boolean LoopCondition evaluates to true, the Statement is executed.	
T2/LJLW/S8	When the Statement finishes executing, the LoopCondition is tested again and the process repeats itself.	
T2/LJLW/S9	This continues until the LoopCondition evaluates to false, in which case the loop immediately bails out.	
T2/LJLW/S10	Because the while loop has no step expression, **it is important to make sure that** || the Statement somehow impacts the LoopCondition.	Rel/Ob
T2/LJLW/S11	Otherwise, **it is possible** || for the loop to repeat infinitely, which is usually a bad thing.	Rel/Ob
T2/LJLW/S12	Following is a simple example of an infinite while loop:	
T2/LJLW/S13	Because the loop condition in this example is permanently set to true, the loop will repeat infinitely, or at least until you manually terminate the programme.	
T2/LJLW/S14	Infinite loops are extremely dangerous because they can result in your computer overheating.	
T2/LJLW/S15	Just kidding!	
T2/LJLW/S16	Actually, infinite loops are useful in some situations.	
T2/LJLW/S17	They are never truly infinite because you can typically terminate one by shutting down the application or applet containing it.	
T2/LJLW/S18	**You can think of** || the while loop as a more general for loop.	Men/Ob
T2/LJLW/S19	To understand what I mean by this, check out the following code;	
T2/LJLW/S20	This is the NASA launch sequence implemented using a while loop instead of a for loop.	
T2/LJLW/S21	Because while loops don't have initialization expressions, the initialization of the counter variable I had to be performed before the loop.	
T2/LJLW/S22	Likewise, the step expression i.. had to be performed within the Statement part of the loop.	
T2/LJLW/S23	Regardless of the structural differences, this while loop is functionally equivalent to the for loop you saw earlier in the chapter.	
T2/LJLW/S24	If a for loop can do everything a while loop can and in a more organized way, then why do we need while loops?	
T2/LJLW/S25	Because there is a time and a place for everything, and in many situations you have no need for initialization and step expressions.	
T2/LJLW/S26	A for loop is overkill in situations like this.	

T2/LJLW/S27	Even more importantly, a while loop is much more readable than a for loop when you have no need for initialization and step expressions.	
T2/LJLW/S28	Consider the following example:	
T2/LJLW/S29	This code demonstrates how a while loop could be used to ask a question and patiently wait for the correct answer.	
T2/LJLW/S30	The loop repeats itself as long as the Boolean variable correct is false.	
T2/LJLW/S31	This results in the code repeating the question as many times as necessary until the user guesses the correct answer.	
T2/LJLW/S32	The details of the methods askQuestion() and isCorrect() aren't important for this example.	
T2/LJLW/S33	Just assume that they somehow present the user with a question, retrieve an answer, and then judge the correctness of the answer.	
T2/LJLW/S34	**The main concern is that** || the isCorrect () method returns a Boolean value that indicates whether or not the answer is correct.	Rel/Prob
T2/LJLW/S35	In this example, **it is impossible** || to know how many times the user will miss the answer and need the question repeated.	Rel/Prob
T2/LJLW/S36	For this reason, the structured step expression of a for loop wouldn't be of much use.	
T2/LJLW/S37	While loops are perfect in situations where you don't know ahead of time how many times a loop needs to be repeated.	
T2/LJLW/S38	If you aren't completely satisfied with while loops, however, there is one other option.	
Title	To do , Or not to do (TDNTD)	
T2/TDNTD/S1	The while loop has a very close relative known as the do loop, or do-while loop, that is surprisingly similar to the while loop.	
T2/TDNTD/S2	Because you're becoming pretty loop savvy, I'll show you the syntax for the do-while loop first and see if you can figure out how it works:	
T2/TDNTD/S3	Give up?	
T2/TDNTD/S4	The do-while loop is basically a while loop with the LoopCondition moved to the end.	
T2/TDNTD/S5	Why is this necessary?	
T2/TDNTD/S6	Because there are some situations where you would like the Statement to execute before evaluating the LoopCondition, instead of afterward.	
T2/TDNTD/S7	This also guarantees that the Statement is executed at least once, regardless of the LoopCondition.	
T2/TDNTD/S8	Let's take a look at the question and answer example implemented using a do-while loop:	
T2/TDNTD/S9	The code really isn't much different than before, except that you no longer need to initialise the correct variable.	
T2/TDNTD/S10	**It is always initially set** || during the first pass through the loop.	Rel/Prob
T2/TDNTD/S11	Although both types of loops accomplish the goal of this example, the do-while loop is a better fit because of its structure more closely mimics the function of the code.	
T2/TDNTD/S12	What do I mean by this?	
T2/TDNTD/S13	Well, if you "read" the code, it is saying "ask the question and if the answer is not correct, ask it again."	
T2/TDNTD/S14	**This makes more sense** || than if it read "if the answer is not correct, ask the question and then check the answer again."	Rel/Ob
T2/TDNTD/S15	Admittedly, this is a subtle difference, but a large part of successful programming is keeping things logical and straightforward.	

T2/TDNTD/S16	You won't always succeed because sometimes code gets complicated regardless of how you construct it, but using loops intelligently is a good start.	
Title	Applet Countdown (AC)	
T2/AC/S1	Have you ever visited a Web page that directed you to another page, but informed you that if you waited a few second sit would automatically take you there?	
T2/AC/S2	I used to run across these pages and wonder how you could make a page wait a few seconds and then automatically navigate to a new page.	
T2/AC/S3	After I started programming in Java, I realised what a trivial task this is.	
T2/AC/S4	In this section you use your knowledge of loops to build a "countdown" applet that counts down from ten to one and then navigates to a new Web page.	
T2/AC/S5	The following figure shows the Countdown applet in action.	
T2/AC/S6	When the applet finishes counting down, it navigates to the web page identified by the page applet parameter.	
T2/AC/S7	As an example, what web site could be better than NASA's to demonstrate how this applet works?	
T2/AC/S8	Following is NASA's Web site, to which the Countdown applet will take you after it finishes its countdown.	
T2/AC/S9	To understand how the Countdown applet works, let's first take a look at the Countdown. Html Web page that contains the embedded applet:	
T2/AC/S10	All this stuff should look pretty familiar to you by now.	
T2/AC/S11	The main thing on which **I want you** || to focus is the page parameter, which is defined as:	Men/Ob
T2/AC/S12	Notice that the value of the page parameter is set to http://www.nasa.gov, which is the URL of NASA's Web site.	
T2/AC/S13	Changing this value enables you to change the page that is loaded after the applet finishes counting down.	
T2/AC/S14	This page could have easily been set as a variable within the applet code, but a recompile would be required to change the page.	
T2/AC/S15	That is the beauty of applet parameters.	
T2/AC/S16	They enable you to customize the function of applets without doing any real programming!	
T2/AC/S17	Let's move on to the actual code required of the countdown applet.	
T2/AC/S18	Unfortunately, the Countdown applet requires some code that is a little beyond the lesson, so I don't expect all of this applet to make sense to you.	
T2/AC/S19	However, you can download the complete source code for the applet from the book's companion Web site, which was mentioned a little earlier in this section.	
T2/AC/S20	Also, the core mechanics of the applet are very straightforward and should be familiar to you from your recent study of loops.	
T2/AC/S21	Following is the run() method in the Countdown applet class, which forms the heart of the applet:	
T2/AC/S22	Ouch, that looks a little messy!	
T2/AC/S23	Try not to get intimidated by any code that doesn't look familiar	
T2/AC/S24	Just concentrate on the loop code.	
T2/AC/S25	As you can see, the for loop counts down from 10 to 1 just like the countdown code you saw earlier in the chapter.	
T2/AC/S26	The Statement part of this for loop is completely new territory, however.	

T2/AC/S27	The call to the repaint () method is necessary to update the applet's window with the new countdown number.	
T2/AC/S28	The call to the thread.sleep() method results in the applet waiting one second, which effectively pauses the countdown for one second between numbers.	
T2/AC/S29	When the for loop finishes, the code gets the page applet parameter and proceed to navigate to the Web page identified by it.	
T2/AC/S30	The code required to navigate to the Web page is probably pretty strange looking to you because it has to deal with exceptions.	
T2/AC/S31	Exceptions are errors caused by unforeseen problems such as your computer running out of memory, your modem coming unplugged, spilling coffee on your keyboard, hurling your monitor out the window, and so on.	
T2/AC/S32	I'll explain exceptions as you encounter them throughout the book.	
T2/AC/S33	The complete source code for the countdown applet follows.	
T2/AC/S34	Although this is a longer program than you are accustomed to seeing, a lot of it should look familiar to you.	
T2/AC/S35	For example, the paint() method code is very similar to the code used in the DateTime applet from Chapter 4, "constructing Applets of Your Own."	
T2/AC/S36	On the other hand, the start() method is entirely new and is related to the applet's use of threads.	
T2/AC/S37	You don't need to understand it fully at this point.	
Title	Breaking Away (BA)	
T2/BA/S1	If you recall from the previous chapter, each case section of a switch branch ends with a break statement.	
T2/BA/S2	Following is an example to recap:	
T2/BA/S3	The purpose of the break statement in this example is to bail out of the switch branch so that no other code is executed.	
T2/BA/S4	The break statement serves a similar purpose in loops.	
T2/BA/S5	It breaks out of a loop regardless of the loop condition.	
T2/BA/S6	Following is an example of circumventing an infinite loop with a break statement:	
T2/BA/S7	Without the assistance of the break statement, this while loop would continue forever thanks to the permanent true loop condition.	
T2/BA/S8	The break statement sidesteps this problem by breaking out of the loop after one hundred iterations (0-99).	
T2/BA/S9	Of course, **it is rare that ||** you would purposely create an infinite loop and then use a break statement to bail out of it.	Rel/Prob
T2/BA/S10	However, the break statement can be very useful in some tricky loops when you need to exit at an otherwise inconvenient time.	
T2/BA/S11	A close relative of the break statement is the continue statement, which is used to skip to the next iteration of a loop.	
T2/BA/S13	The following example shows how a continue statement can be used to print only the even numbers between 1 and 100:	
T2/BA/S14	Having trouble seeing how this one works?	
T2/BA/S15	Think back to the modulus operator (%), which returns the reminder of a division.	
T2/BA/S16	Now consider what the remainder of a division by 2 yields for even and odd numbers.	
T2/BA/S17	Aha!	
T2/BA/S18	Even numbers divided by 2 always yield a remainder of 0, and odd numbers always leave a remainder of 1!	
T2/BA/S19	The example code exploits this characteristic of even and odd numbers to skip to the next iteration bypasses the println()	

	call, which prevents odd numbers from being printed.	
T2/BA/S20	Pretty tricky!	
Title	The Least you need to know (LNK)	
T2/LNK/S1	Computers are often called upon to perform tasks we humans find to be utterly redundant.	
T2/LNK/S2	As dull as some humans can be, **I guarantee you** || computers are much duller when it comes to repeating the same thing over and over.	Men/Ob
T2/LNK/S3	Java enables you to build programs that repeat themselves through the use of loops.	
T2/LNK/S4	The different types of loops basically perform the same function.	
T2/LNK/S5	They repeat a section of code over and over.	
T2/LNK/S6	Let's go over the main points you learned about loops in this chapter:	
T2/LNK/S7	Loops can execute as many statements as you want them to, provided the statements are grouped together as a single compound statement enclosed by curly braces ({}).	
T2/LNK/S8	A for loop is used to repeat a section of code a given number of iterations.	
T2/LNK/S9	A while loop is a more general for loop.	
T2/LNK/S10	A do while is a while loop with the loop condition moved to the end.	
T2/LNK/S11	The break statement is used to break out of a loop regardless of the loop condition.	
T2/LNK/S12	The continue statement is used to skip to the next iteration of a loop.	

Appendix 8A

Text 1 – Mental Projection Clauses with 'Probability' or 'Obligation' Type

Keys:
1. Men/Prob denotes Mental Projection clause with Probability
2. Men/Ob denotes Mental Projection clause with Obligation

Note:
1. Mental projection clauses are in bold and separated with oblique lines (||).

Eg.	Label	Sentence	Code
1	T1/MNE/S10	From this example, **you can see that** || a control variable can be a *float* type.	Men/Prob
2	T1/MNE/S15	However, **you will be stunned to see that** || the result is actually *49.50000000000003.*	Men/Prob
3	T1/CS/S3	If you can write programs using *loops*, **you know** || how to program!	Men/Prob
4	T1/FGCD/S6	**You know that** || number 1 is a common divisor, but it may not be the greatest common divisor.	Men/Prob
5	T1/FGCD/S28	**You might think that** || a divisor for a number *n1* cannot be greater than *n1 / 2.*	Men/Prob

Appendix 8B

Text 2 – Mental Projection Clauses with 'Probability' or 'Obligation' Type

Keys:
1. Men/Prob denotes Mental Projection clause with Probability
2. Men/Ob denotes Mental Projection clause with Obligation

Note:
1. Mental projection clauses are in bold and separated with oblique lines (||).

Eg.	Label	Mental Projection Clauses	Code
1	T2/LEO/S2	**I mean** ||, you keep listening, and they keep talking, and it all sounds the same.	Men/Ob
2	T2/LEO/S3	And they talk somemore and you listen somemore and **you wonder** || if it will ever end!	Men/Prob
3	T2/LEO/S4	Congratulations, **you just experienced** || a perfect example of a verbal *loop*!	Men/Prob
4	T2/GRL/S9	**You probably wonder exactly** || how the *loop* code works.	Men/Prob
5	T2/GRL/S26	Whew, **that explanation seemed** || a little long-winded, and that's coming from the person that wrote it!	Men/Prob
6	T2/GRL/S30	**You can feel safe and secure knowing that** || I'm not running for president or trying to help you visualize my answer to global trade.	Men/Prob
7	T2/GRL/S31	**I just want** || to help you learn how *loops* work!	Men/Ob
8	T2/LJLW/S18	**You can think** || of the *while loop* as a more general for *loop*.	Men/Ob
9	T2/AC/S2	**I** used to run across these pages and **wonder how** || you could make a page wait a few seconds and then automatically navigate to a new page.	Men/Prob
10	T2/AC/S3	After I started programming in *Java*, **I realised** || what a trivial task this is.	Men/Ob
11	T2/AC/S11	The main thing on which **I want you** || to focus is the page parameter, which is defined as:	Men/Ob
12	T2/LNK/S2	As dull as some humans can be, **I guarantee you** || computers are much duller when it comes to repeating the same thing over and over.	Men/Ob

Appendix 9A

Text 1 – Relational Projection Clauses with 'Probability' or 'Obligation' Type

Keys:
1. Rel/Prob denotes Relational projection clause with Probability
2. Rel/Ob denotes Relational projection clause with Obligation

Note:
1. Relational projection clauses are in bold and separated with oblique lines (||).

Eg.	Label	Relational Projection Clauses	Code
1	T1/INT/S2	**It would be tedious** || to have to write the following *statement* a hundred times.	Rel/Prob
2	T1/WL/S6	**It is always evaluated ||** before the *loop* body is executed.	Rel/Ob
3	T1/FL/S26	**It is good programming practice** || to declare it in the *initial-action* of the *for loop*.	Rel/Prob
4	T1/MNE/S19	**The fundamental problem is that** || the *floating-point* numbers are represented by approximation.	Rel/Prob
5	T1/FGCD/S18	**It is important** || to think before you type.	Rel/Prob
6	T1/FSA/S10	**This suggests that** || you can try to find the *salesAmount* to match a given commission through incremental approximation.	Rel/Prob
7	T1/FSA/S14	This is a tedious job for humans, but **it is exactly** || what a computer is good for.	Rel/Prob
8	T1/KBC/S3	**It is generally used** || with an *if statement*.	Rel/Ob
9	T1/KBC/S6	**This keyword is generally used** || with an *if statement*.	Rel/Ob

Appendix 9B

Text 2 – Relational Projection Clauses with 'Probability' or 'Obligation' Type

Keys:
1. Rel/Prob denotes Relational projection clause with Probability
2. Rel/Ob denotes Relational projection clause with Obligation

Note:
1. Relational projection clauses are in bold and separated with oblique lines (||).

Eg	Label	Relational Projection Clauses	Code
1	T2/LEO/S1	Have you ever been talking to someone and **it seems like** || he or she is saying the same thing over and over?	Rel/Prob
2	T2/GRL/S27	Unfortunately, **it isn't always easy** || to verbalize the flow of program code.	Rel/Prob
3	T2/GRL/S28	**This is why it's easy** || to fall back on figures.	Rel/Prob
4	T2/GRL/S41	**This is a very popular way** || to handle arrays.	Rel/Prob
5	T2/GRL/S42	**It is necessary ||** to subtract 1 in this case because all Java array indexes start with 0, which means they are zero based.	Rel/Prob
6	T2/GRL/S43	**It might be worth noting that** || although zero-based arrays were used in other programming languages in the 1980s and before, they have nothing to do with the 80s movie Less than Zero or the 80s hit song saved by Zero.	Rel/Ob
7	T2/LJLW/S10	Because the while *loop* has no step expression, **it is important to make sure that** || the *Statement* somehow impacts the LoopCondition.	Rel/Ob
8	T2/LJLW/S11	Otherwise, **it is possible** || for the *loop* to repeat infinitely, which is usually a bad thing.	Rel/Ob
9	T2/LJLW/S34	**The main concern is that** || the *isCorrect () method* returns a Boolean value that indicates whether or not the answer is correct.	Rel/Prob
10	T2/LJLW/S35	In this example, **it is impossible** || to know how many times the user will miss the answer and need the question repeated.	Rel/Prob
11	T2/TDNTD/S10	**It is always initially set** || during the first pass through the *loop*.	Rel/Prob
12	T2/TDNTD/S14	**This makes more sense** || than if it read "if the answer is not correct, ask the question and then check the answer again."	Rel/Ob
13	T2/BA/S9	Of course, **it is rare that ||** you would purposely create an infinite *loop* and then use a *break statement* to bail out of it.	Rel/Prob

Appendix 10A

Text 1 – Features of Metaphor of Mood and Metaphor Modality

Notes:
1. For every table entry with metaphorical clauses with Metaphor of Mood, the metaphorical sentence precedes the congruent sentence.
2. Mental projection clauses are in bold and separated with oblique lines (||).
3. Relational projection clauses are in bold and separated with oblique lines (||).

Keys:
1. Dec/St/Co denotes Declarative clauses realizing Statement and Command
2. Int/Qu/St denotes Interrogative clauses realizing Question and Statement
3. Int/Qu/Co denotes Interrogative clauses realizing Question and Command
4. Imp/Co/St denotes Imperative clauses realizing Command and Statement
5. Men/Prob denotes Mental Projection clause with Probability
6. Men/Ob denotes Mental Projection clause with Obligation
7. Rel/Prob denotes Relational projection clause with Probability

4.1	INTRODUCTION	Code
T1/INT/S1	Suppose that you need to *print* a *string* (e.g., "Welcome to *Java*!") a hundred times. What if you need to *print* a *string* (e.g., "Welcome to *Java*!") a hundred times?	Dec/St/Qu
T1/INT/S2	**It would be tedious** || to have to write the following *statement* a hundred times.	Rel/Prob
T1/INT/S3	*Java* provides a powerful control structure called a *loop* that controls how many times an operation or a sequence of operations is performed in succession.	
T1/INT/S4	Using a *loop statement*, you simply tell the computer to *print* a *string* a hundred times without having to code the *print statement* a hundred times. Using a *loop statement*, tell the computer to *print* a *string* a hundred times without having to code the *print statement* a hundred times.	Dec/St/Co
T1/INT/S5	*Loops* are structures that control repeated executions of a block of *statements*.	
T1/INT/S6	The concept of *looping* is fundamental to programming.	
T1/INT/S7	*Java* provides three types of *loop statements*: *while loops*, *do-while loops*, and *for loops*.	
4.2	THE *WHILE LOOP*	
T1/WL/S1	The syntax *for* the *while loop* is as follows.	
T1/WL/S2	The *while loop* flow chart is shown in *Figure 4.1(a)*.	
T1/WL/S3	The part of the *loop* that contains the *statements* to be repeated is called the *loop body*.	
T1/WL/S4	A one-time execution of a *loop* body is referred to as an iteration of the *loop*.	

T1/WL/S5	Each *loop* contains a *loop-continuation* condition, a Boolean expression that controls the execution of the body.	
T1/WL/S6	**It is always evaluated** || b*e*fo*r*e the *loop* body is executed.	Rel/Ob
T1/WL/S7	If its evaluation is true, the *loop* body is executed.	
T1/WL/S8	If its evaluation is false, the entire *loop* terminates and the program control turns to the *statement* that follows the *while loop*.	
T1/WL/S9	For example, the following *while loop* prints "Welcome to *Java*!" a hundred times.	
T1/WL/S10	The flow chart of the preceding *statement* is shown in *Figure 4.1(b).*	
T1/WL/S11	The variable *count* is initially 0.	
T1/WL/S12	The *loop* checks whether (*count* < *100*) is true.	
T1/WL/S13	If so, it executes the *loop* body to *print* the message "Welcome to *Java*!" and increments *count* by 1.	
T1/WL/S14	It repeatedly executes the *loop* body until (*count* < *100*) becomes false.	
T1/WL/S15	When (*count* < *100*) is false (i.e., when *count* reaches *100*), the *loop* terminates and the next *statement* after the *loop statement* is executed.	
4.2.1	AN ADVANCED MATH LEARNING TOOL	
T1/AMLT/S1	The Math subtraction learning tool program in *Listing* 3.5, *SubtractionTutor.Java*, generates just one question for each run.	
T1/AMLT/S2	You can use a *loop* to generate questions repeatedly. Use a *loop* to generate questions repeatedly.	
T1/AMLT/S3	*Listing* 4.1 gives a program that generates ten questions and reports the number of correct answers after a student answers all ten questions.	
T1/AMLT/S4	The program also displays the time spent on the test and lists all the questions, as shown in *Figure 4.2*.	
T1/AMLT/S5	The program uses the control variable *count* to control the execution of the *loop*.	
T1/AMLT/S6	*Count* is initially 0 (*line* 6) and is increased by 1 in each iteration (*line* 39).	
T1/AMLT/S7	A subtraction question is displayed and processed in each iteration.	
T1/AMLT/S8	The program obtains the time before the test starts in *line* 7 and the time after the test ends in *line* 45, and computes the test time in *line* 46.	
T1/AMLT/S9	The test time is in milliseconds and is converted to seconds in *line 50.*	
4.2.2	CONTROLLING A *LOOP* WITH A CONFIRMATION DIALOG	
T1/LCD/S1	The preceding example executes the *loop* ten times.	
T1/LCD/S2	If you want the user to decide whether to take another question, you can use a confirmation dialog to control the *loop*. If you want the user to decide whether to take another question, use a confirmation dialog to control the *loop*.	Dec/St/Co
T1/LCD/S3	A confirmation dialog can be created using the following *statement*. Create a confirmation dialog using the following *statement.*	Dec/St/Co
T1/LCD/S4	When a button is clicked, the *method* returns an option value.	
T1/LCD/S5	The value is *JOptionPane.YES_OPTION (0)* for the Yes button, *JOptionPane.NO_OPTION (1)* for the No button, and *JOptionPane.CANCEL_OPTION (2)* for the Cancel button.	
T1/LCD/S6	For example, the following *loop* continues to execute until the user clicks the *No* or *Cancel* button.	
T1/LCD/S7	You can rewrite *Listing 4.1* using a confirmation dialog to let the user decide whether to continue the next	Dec/St/Co

	question. Rewrite *Listing 4.1* using a confirmation dialog to let the user decide whether to continue the next question.	
4.2.3	CONTROLLING A *LOOP* WITH A SENTINEL VALUE	
T1/LSV/S1	Another common technique *for* controlling a *loop* is to designate a special value when reading and processing a set of values.	
T1/LSV/S2	This special input value, known as a *sentinel value*, signifies the end of the *loop*.	
T1/LSV/S3	*Listing 4.2* writes a program that reads and calculates the *sum* of an unspecified number of integers.	
T1/LSV/S4	The input 0 signifies the end of the input.	
T1/LSV/S5	Do you need to declare a new variable for each input value?	Int/Qu/St
T1/LSV/S6	No. There is no need to declare a new variable for each input.	
T1/LSV/S7	Just use one variable named *data* (*line* 9) to store the input value and use a variable named *sum* (*line* 12) to store the total. One variable named *data* (*line* 9) is used to store the input value and a variable named *sum* (*line* 12) is used to store the total	Imp/Co/St
T1/LSV/S8	Whenever a value is read, assign it to *data* and added to *sum* (*line* 14) if it is not zero.	
T1/LSV/S9	A sample run of the program is shown in *Figure 4.3*. See *Figure 4.3* for a sample run of the program	Dec/St/Co
T1/LSV/S10	If *data* is not 0, it is added to the *sum* (*line* 14) and the next items of input data are read (*line*s 12–19).	
T1/LSV/S11	If data is 0, the *loop* body is no longer executed and the *while loop* terminates.	
T1/LSV/S12	The input value 0 is the sentinel value for this *loop*.	
T1/LSV/S13	Note that if the first input read is 0, the *loop* body never executes, and the resulting *sum* is 0. If the first input read is 0, the *loop* body never executes, and the resulting *sum* is 0.	Imp/Co/St
T1/LSV/S14	The program uses a *while loop* to add an unspecified number of integers.	
4.3	THE *DO-WHILE LOOP*	
T1/DWL/S1	The *do-while loop* is a variation of the *while loop*.	
T1/DWL/S2	Its syntax is given below: See below for its syntax:	Dec/St/Co
T1/DWL/S3	Its execution flow chart is shown in *Figure 4.4*. See *Figure 4.4* for its execution flow chart.	Dec/St/Co
T1/DWL/S4	The *loop* body is executed first.	
T1/DWL/S5	Then the *loop-continuation-condition* is evaluated.	
T1/DWL/S6	If the evaluation is true, the *loop* body is executed again.	
T1/DWL/S7	If it is false, the *do-while loop* terminates.	
T1/DWL/S8	The major difference between a *while loop* and a *do-while loop* is the order in which the *loop-continuation-condition* is evaluated and the *loop* body executed.	
T1/DWL/S9	The *while loop* and the *do-while loop* have equal expressive power.	
T1/DWL/S10	Sometimes one is a more convenient choice than the other.	
T1/DWL/S11	For example, you can rewrite the *while loop* in *Listing 4.2* using a *do-while loop*, as shown in *Listing 4.3*.	Dec/St/Co

	For example, rewrite the *while loop* in *Listing 4.2* using a *do-while loop*, as shown in *Listing 4.3*.	
4.4	THE *FOR LOOP*	
T1/FL/S1	Often you write a *loop* in the following common form.	
T1/FL/S2	A *for loop* can be used to simplify the preceding *loop*. Use a *for loop* to simplify the preceding *loop*.	Dec/St/Co
T1/FL/S3	In general, the syntax of a *for loop* is as shown below. In general, see below for the syntax of a *for loop.*	Dec/St/Co
T1/FL/S4	The flow chart of the *for loop* is shown in *Figure 4.5(a)*. See *Figure 4.5(a)* for the flow chart of the *for loop.*	
T1/FL/S5	The *for loop statement* starts with the keyword *for*, followed by a pair of parentheses enclosing *initial-action*, *loop-continuation-condition*, and *action-after-each-iteration*, and followed by the *loop* body enclosed inside braces. Start the *for loop statement* with the keyword *for,* enclose *initial-action, loop-continuation-condition* with a pair of parenthesis, and enclose the *loop* body inside braces.	Dec/St/Co
T1/FL/S6	*Initial-action*, *loop-continuation-condition*, and *action-after-each-iteration* are separated by semicolons. Separate *Initial-action*, *loop-continuation-condition*, and *action-after-each-iteration* with semicolons.	Dec/St/Co
T1/FL/S7	A *for loop* generally uses a variable to control how many times the *loop* body is executed and when the *loop* terminates. Use a variable in a *for loop* to control how many times the *loop* body is executed and when the *loop* terminates.	Dec/St/Co
T1/FL/S8	This variable is referred to as a control variable. Refer this variable as a control variable.	Dec/St/Co
T1/FL/S9	The *initial-action* often initializes a control variable, the *action-after-each-iteration* usually increments or decrements the control variable, and the *loop-continuation-condition* tests whether the control variable has reached a termination value.	
T1/FL/S10	For example, the following *for loop* prints "Welcome to *Java*!" a hundred times. See the following to understand how *for loop* prints "Welcome to *Java*!" a hundred times.	Dec/St/Co
T1/FL/S11	The flow chart of the *statement* is shown in *Figure 4.5(b)*. See *Figure 4.5(b)* for the flow chart of the *statement.*	Dec/St/Co
T1/FL/S12	The *for loop* initializes *i* to 0, then repeatedly executes the *println statement* and evaluates *i*++ *while i* is less than *100.*	
T1/FL/S13	The *initial-action, i=0,* initializes the control variable, *i.*	
T1/FL/S14	The *loop-continuation-condition*, *i< 100* is a Boolean expression.	
T1/FL/S15	The expression is evaluated at the beginning of each iteration.	
T1/FL/S16	If this condition is true, execute the *loop* body.	
T1/FL/S17	If it is false, the *loop* terminates and the program control turns to the *line* following the *loop*.	
T1/FL/S18	The *action-after-each-iteration, i++,* is a *statement* that adjusts the control variable.	
T1/FL/S19	This *statement* is executed after each iteration.	
T1/FL/S20	It increments the control variable.	
T1/FL/S21	Eventually, the value of the control variable should force the *loop-continuation-condition* to become false.	

T1/FL/S22	Otherwise the *loop* is infinite.	
T1/FL/S23	If there is only one *statement* in the *loop* body, as in this example, the braces can be omitted. If there is only one *statement* in the *loop* body, as in this example, omit the braces.	Dec/St/Co
T1/FL/S24	The control variable must always be declared inside the control structure of the *loop* or before the *loop*. Declare the control variable inside the control structure of the *loop* or before the *loop*.	Dec/St/Co
T1/FL/S25	The *loop* control variable is used only in the *loop*, and not elsewhere. Use the *loop* control variable only in the *loop*, and not elsewhere.	Dec/St/Co
T1/FL/S26	**It is good programming practice** || to declare it in the *initial-action* of the *for loop*.	Rel/Prob
T1/FL/S27	If the variable is declared inside the *loop* control structure, it cannot be referenced outside the *loop*.	
T1/FL/S28	For example, you cannot reference *i* outside the *for loop* in the preceding code, because it is declared inside the *for loop*. Do not reference *i* outside the *for loop* in the preceding code, because it is declared inside the *for loop*.	Dec/St/Co
4.5.	WHICH *LOOP* TO USE?	
T1/WLU/S1	The *while loop* and *for loop* are called pre-test *loops* because the continuation condition is checked be*fo*re the *loop* body is executed. Refer the *while loop* and *for loop as* pre-test *loops* because the continuation condition is checked be*fo*re the *loop* body is executed.	Dec/St/Co
T1/WLU/S2	The *do-while loop* is called a post-test *loop* because the condition is checked after the *loop* body is executed. Refer The *do-while loop as* a post-test *loop* because the condition is checked after the *loop* body is executed	Dec/St/Co
T1/WLU/S3	The three *fo*rms of *loop statements*, *while*, *do-while*, and *for*, are expressively equivalent.	
T1/WLU/S4	That is, you can write a *loop* in any of these three forms. Write a *loop* in any of these three forms.	Dec/St/Co
T1/WLU/S5	For example, a *while loop* in (a) in the following figure can always be converted into the *for loop* in (b): See the following figure on how , a *while loop* in (a) can always be converted into the *for loop* in (b):	Dec/St/Co
T1/WLU/S6	A *for loop* in (a) in the next figure can generally be converted into the *while loop* in (b) except in certain special cases (see Review Question 4.12 *for* such a case): See the next figure on how A *for loop* in (a) can generally be converted into the *while loop* in (b) except in certain special cases (see Review Question 4.12 *for* such a case):	Dec/St/Co
T1/WLU/S7	Use the *loop statement* that is most intuitive and comfortable for you. You can use the *loop statement* that is most intuitive and comfortable for you.	Imp/Co/St
T1/WLU/S8	In general, a *for loop* may be used if the number of repetitions is known, as, for example, when you need to *print* a message a hundred times. In general, use a *for loop* used if the number of repetitions is known, as, for example, when you need to *print* a message a hundred times.	Dec/St/Co
T1/WLU/S9	A *while loop* may be used if the number of repetitions is not known, as in the case of reading the numbers until the input is 0. Use a *while loop* if the number of repetitions is not known, as in the case of reading the numbers until the input is 0.	Dec/St/Co
T1/WLU/S10	A *do-while loop* can be used to replace a *while loop* if the *loop* body has to be executed before the continuation	Dec/St/Co

	condition is tested. Use a *do-while loop* to replace a *while loop* if the *loop* body has to be executed before the continuation condition is tested.	
4.6.	*NESTED LOOPS*	
T1/NL/S1	*Nested loops* consist of an outer *loop* and one or more inner *loops*.	
T1/NL/S2	Each time the outer *loop* is repeated, the inner *loops* are reentered, and started anew.	
T1/NL/S3	*Listing 4.4* presents a program that uses *nested for loops* to *print* a multiplication table, as shown in *Figure 4.6*. See Listing 4.4 that presents a program that uses *nested for loops* to *print* a multiplication table, as shown in *Figure 4.6*	Dec/St/Co
T1/NL/S4	The program displays a title (*line* 7) on the first *line* and dashes (-) (*line* 8) on the second *line*.	
T1/NL/S5	The first *for loop* (*line*s 12–13) displays the numbers 1 through 9 on the third *line*.	
T1/NL/S6	The next *loop* (*line*s 18–28) is a *nested for loop* with the control variable *i* in the outer *loop* and *j* in the inner *loop*.	
T1/NL/S7	For each *i,* the product *i * j* is displayed on a *line* in the inner *loop*, with *j* being *1, 2, 3, …, 9.*	
T1/NL/S8	The *if statement* in the inner *loop* (*line*s 22–25) is used so that the product will be aligned properly.	
T1/NL/S9	If the product is a single digit, it is displayed with an extra space before it.	
4.7	MINIMIZING NUMERICAL ERRORS	
T1/MNE/S1	Numeric errors involving floating-point numbers are inevitable.	
T1/MNE/S2	This section discusses how to minimize such errors through an example. Read this section that discusses how to minimize such errors through an example.	Dec/St/Co
T1/MNE/S3	*Listing 4.5* presents an example that *sum*s a series that starts with *0.01* and ends with *1.0.* See *Listing 4.5* that presents an example that *sum*s a series that starts with *0.01* and ends with *1.0.*	Dec/St/Co
T1/MNE/S4	The numbers in the series will increment by *0.01*, as follows: *0.01 + 0.02 + 0.03* and so on.	
T1/MNE/S5	The output of the program appears in *Figure 4.7.* See the output of the program in *Figure 4.7*. Dec/St/Co	
T1/MNE/S6	The *for loop* (*lines 9–10*) repeatedly adds the control variable *i* to the *sum.*	
T1/MNE/S7	This variable, which begins with *0.01*, is incremented by *0.01* after each iteration.	
T1/MNE/S8	The *loop* terminates when *i* exceeds *1.0.*	
T1/MNE/S9	The *for loop initial action* can be any *statement*, but it is often used to initialize a control variable.	
T1/MNE/S10	From this example, **you can see that** || a control variable can be a *float* type. From this example, see that a control variable can be a *float* type.	Dec/St/Co & Men/Prob
T1/MNE/S11	In fact, it can be any data type.	
T1/MNE/S12	The exact *sum* should be *50.50,* but the answer is *50.499985.*	
T1/MNE/S13	The result is not precise because computers use a fixed number of bits to represent floating-point numbers, and thus cannot represent some *floating*-point numbers exactly.	
T1/MNE/S14	If you change *float* in the program to *double* as follows, you should see a slight improvement in precision because a *double* variable takes sixty-four *bits*, whereas a *float* variable takes thirty-two *bits*. If you change *float* in the program to *double* as follows, notice a slight improvement in precision because a *double* variable takes sixty-four *bits*, whereas a *float* variable takes thirty-two *bits*.	Dec/St/Co

T1/MNE/S15	However, **you will be stunned to see that** || the result is actually *49.50000000000003.* However, see that the result is actually *49.50000000000003.*	Dec/St/Co & Men/Prob
T1/MNE/S16	What went wrong? There is a mistake.	Int/Qu/St
T1/MNE/S17	If you *print* out *i* for each iteration in the *loop*, you will see that the last *i* is slightly larger than 1 (not exactly 1).	
T1/MNE/S18	This causes the last *i* not to be added into *sum*.	
T1/MNE/S19	**The fundamental problem is that** || the *floating-point* numbers are represented by approximation.	Rel/Prob
T1/MNE/S20	Errors commonly occur.	
T1/MNE/S21	There are two ways to fix the problem.	
T1/MNE/S22	Minimizing errors by processing large numbers first. Minimize errors by processing large numbers first.	Dec/St/Co
T1/MNE/S23	Using an integer *count* to ensure that all the numbers are processed. Use an integer *count* to ensure that all the numbers are processed.	Dec/St/Co
T1/MNE/S24	To minimize errors, add numbers from *1.0, 0.99,* down to *0.1,* as follows: To minimize errors, you can add numbers from *1.0, 0.99,* down to *0.1,* as follows:	Imp/Co/St
T1/MNE/S25	To ensure that all the items are added to *sum*, use an integer variable to *count* the items. To minimize errors, you can use an integer variable to *count* the items.	Imp/Co/St
T1/MNE/S26	Here is the new *loop*. See here for the new *loop.*	Dec/St/Co
T1/MNE/S27	After this *loop*, *sum* is *50.50000000000003.*	
4.8	CASE STUDIES	
T1/CS/S1	Control *statements* are fundamental in programming.	
T1/CS/S2	The ability to write control *statements* is essential in learning *Java* programming.	
T1/CS/S3	If you can write programs using *loops*, **you know** || how to program!	Men/Prob
T1/CS/S4	For this reason, this section presents three additional examples of how to solve problems using *loops*. For this reason, read this section that presents three additional examples of how to solve problems using *loops*.	Dec/St/Co
4.8.1	EXAMPLE: FINDING THE GREATEST COMMON DIVISOR	
T1/FGCD/S1	This section presents a program that prompts the user to enter two positive integers and finds their greatest common divisor. Read this section that presents a program that prompts the user to enter two positive integers and finds their greatest common divisor.	Dec/St/Co
T1/FGCD/S2	The greatest common divisor of two integers *4* and *2* is *2.*	
T1/FGCD/S3	The greatest common divisor of two integers *16* and *24* is *8.*	
T1/FGCD/S4	How do you find the greatest common divisor? Find the greatest common divisor	Dec/St/Co
T1/FGCD/S5	Let the two input integers be *n1* and *n2.* You can let the two input integers be *n1* and *n2.*	Imp/Co/St
T1/FGCD/S6	**You know that** || number 1 is a common divisor, but it may not be the greatest common divisor.	Men/Prob
T1/FGCD/S7	So you can check whether *k (*for *k = 2, 3, 4* and so on) is a common divisor for *n1* and *n2*, until *k* is greater than	Dec/St/Co

	n1 or *n2.* Check whether *k (*for *k = 2, 3, 4* and so on) is a common divisor for *n1* and *n2*, until *k* is greater than *n1* or *n2.*	
T1/FGCD/S8	Store the common divisor in a variable named *gcd.* You can store the common divisor in a variable named *gcd.*	Imp/Co/St
T1/FGCD/S9	Initially, *gcd* is *1.*	
T1/FGCD/S10	Whenever a new common divisor is found, it becomes the new *gcd.*	
T1/FGCD/S11	When you have checked all the possible common divisors from *2* up to *n1* or *n2*, the value in variable *gcd* is the greatest common divisor.	Dec/St/Co
T1/FGCD/S12	The idea can be translated into the following *loop*: Translate the idea into the following *loop*	Dec/St/Co
T1/FGCD/S13	The complete program is given in *Listing* 4.6, and a sample run of the program is shown in *Figure 4.8*. See the complete program in *Listing 4.6* and a sample run of the program in *Figure 4.8.*	
T1/FGCD/S14	The program finds the greatest common divisor for two integers.	
T1/FGCD/S15	How did you write this program? Describe how you wrote this program.	Dec/St/Co
T1/FGCD/S16	Did you immediately begin to write the code?	
T1/FGCD/S17	No. You didn't immediately begin to write the code.	Int/Qu/St
T1/FGCD/S18	**It is important** || to think before you type. Think before you type.	Dec/St/Co & Rel/Prob
T1/FGCD/S19	Thinking enables you to generate a logical solution for the problem without concern about how to write the code.	
T1/FGCD/S20	Once you have a logical solution, type the code to translate the solution into a *Java* program. Once you have a logical solution, you can type the code to translate the solution into a *Java* program.	Imp/Co/St
T1/FGCD/S21	The translation is not unique.	
T1/FGCD/S22	For example, you could use a *for loop* to rewrite the code as follows: For example, use a *for loop* to rewrite the code as follows:	Dec/St/Co
T1/FGCD/S23	A problem often has multiple solutions.	
T1/FGCD/S24	The *GCD* problem can be solved in many ways. You can solve The *GCD* problem in many ways.	
T1/FGCD/S25	*Exercise 4.15* suggests another solution. See *Exercise 4.15* that suggests another solution.	
T1/FGCD/S26	A more efficient solution is to use the classic Euclidean algorithm. For a more efficient solution, use the classic Euclidean algorithm.	Dec/St/Co
T1/FGCD/S27	See *http://www.cut-the-knot.org/blue/Euclid.shtml* for more information. You can see *http://www.cut-the-knot.org/blue/Euclid.shtml* for more information.	Imp/Co/St
T1/FGCD/S28	**You might think that** || a divisor for a number *n1* cannot be greater than *n1 / 2.*	Men/Prob
T1/FGCD/S29	So you would attempt to improve the program using the following *loop*:	
T1/FGCD/S30	This revision is wrong.	

T1/FGCD/S31	Can you find the reason? Find the reason.	Int/Qu/Co
T1/FGCD/S32	See *Review Question* 4.9 for the answer. You can see *Review Question* 4.9 for the answer.	Imp/Co/St
4.8.2	EXAMPLE: FINDING THE SALES AMOUNT	
T1/FSA/S1	You have just started a sales job in a department store.	
T1/FSA/S2	Your pay consists of a base salary and a commission.	
T1/FSA/S3	The base salary is *$5,000.*	
T1/FSA/S4	The scheme shown below is used to determine the commission rate.	
T1/FSA/S5	Your goal is to earn *$30,000* a year.	
T1/FSA/S6	This section writes a program that finds the minimum amount of sales you have to generate in order to make *$30,000.* Read this section that writes a program that finds the minimum amount of sales you have to generate in order to make *$30,000.*	Dec/St/Co
T1/FSA/S7	Since your base salary is *$5,000*, you have to make *$25,000* in commissions to earn *$30,000* a year.	
T1/FSA/S8	What is the sales amount *for* a *$25,000* commission? Find the sales amount *for* a *$25,000* commission.	Dec/St/Co
T1/FSA/S9	If you know the sales amount, the commission can be computed as follows. If you know the sales amount, compute the commission as follows.	Dec/St/Co
T1/FSA/S10	**This suggests that** || you can try to find the *salesAmount* to match a given commission through incremental approximation. Try to find the *salesAmount* to match a given commission through incremental approximation.	Dec/St/Co & Rel/Prob
T1/FSA/S11	For a *salesAmount* of *$0.01 (1 cent),* find commission.	
T1/FSA/S12	If commission is less than *$25,000*, increment *salesAmount* by *0.01* and find commission again.	
T1/FSA/S13	If commission is still less than *$25,000,* repeat the process until it is greater than or equal to *$25,000.*	
T1/FSA/S14	This is a tedious job for humans, but **it is exactly what** || a computer is good for.	Rel/Prob
T1/FSA/S15	You can write a *loop* and let a computer execute it painlessly. Write a *loop* and let a computer execute it painlessly.	Dec/St/Co
T1/FSA/S16	The idea can be translated into the following *loop*: Translate the idea into the following loop:	Dec/St/Co
T1/FSA/S17	The complete program is given in *Listing 4.7*, and a sample run of the program is shown in *Figure 4.9*. Read the complete program given in *Listing 4.7*, and a sample run of the program in *Figure 4.9.*	
T1/FSA/S18	The *do-while loop* (*line*s 12–24) is used to repeatedly compute commission for an incremental *salesAmount.*	
T1/FSA/S19	The *loop* terminates when commission is greater than or equal to a constant *COMMISSION_SOUGHT.*	
T1/FSA/S20	In *Exercise 4.17*, you will rewrite this program to let the user enter *COMMISSION_SOUGHT* dynamically from an input dialog. In *Exercise 4.17*, rewrite this program to let the user enter *COMMISSION_SOUGHT* dynamically from an input dialog.	Dec/St/Co
T1/FSA/S21	You can improve the performance of this program by estimating a higher *INITIAL_SALES_AMOUNT (e.g.,*	Dec/St/Co

	25000). Improve the performance of this program by estimating a higher *INITIAL_SALES_AMOUNT (e.g., 25000).*	
T1/FSA/S22	What is wrong if *salesAmount* is incremented after the commission is computed, as follows? Find the mistake if *salesAmount* is incremented after the commission is computed, as follows?	Dec/St/Co
T1/FSA/S23	The change is erroneous because *salesAmount* is 1 cent more than is needed for the commission when the *loop* ends.	
T1/FSA/S24	This is a common error in *loops*, known as the *off-by-one error*.	
4.8.3	EXAMPLE: DISPLAYING A PYRAMID OF NUMBERS	Dec/St/Co
T1/DPN/S1	This section presents a program that prompts the user to enter an integer from *1* to *15* and displays a pyramid. Read this section that presents a program that prompts the user to enter an integer from *1* to *15* and displays a pyramid.	Dec/St/Co
T1/DPN/S2	If the input integer is *12*, for example, the output is shown in *Figure 4.10*. If the input integer is *12*, for example, see the output in *Figure 4.10.*	
T1/DPN/S3	The program uses *nested loops* to *print* numbers in a triangular pattern.	
T1/DPN/S4	Your program receives the input *for* an integer *(numberOfLines)* that represents the total number of lines.	
T1/DPN/S5	It displays all the lines one by one.	
T1/DPN/S6	Each line has three parts.	
T1/DPN/S7	The first part comprises the spaces before the numbers;	
T1/DPN/S8	The second part, the leading numbers, such as 3 2 1 in *line* 3.	
T1/DPN/S9	And the last part, the ending numbers, such as 2 3 in *line* 3.	
T1/DPN/S10	Each number occupies three spaces.	
T1/DPN/S11	Display an empty space before a double-digit number, and display two empty spaces before a single-digit number.	
T1/DPN/S12	You can use an outer *loop* to control the lines. Use an outer *loop* to control the lines.	Dec/St/Co
T1/DPN/S13	At the n^{th} row, there are *(numberOfLines – n)*3* leading spaces, the leading numbers are *n, n-1, … 1,* and the ending numbers are *2, ..., n.*	
T1/DPN/S14	You can use three separate inner *loops* to *print* each part. Use three separate inner *loops* to *print* each part.	Dec/St/Co
T1/DPN/S15	Here is the algorithm for the problem. Read the algorithm for the problem.	Dec/St/Co
T1/DPN/S16	The complete program is given in *Listing 4.8*. See the complete program in *Listing 4.8.*	Dec/St/Co
T1/DPN/S17	The program uses the *print method (lines 20, 24, and 28)* to display a *string* to the console.	
T1/DPN/S18	The conditional expression *(num >= 10) ? " " + num : " " + num* in *line*s 24 and 28 returns a *string* with a single empty space before the number if the number is greater than or equal to 10, and otherwise returns a *string* with two empty spaces before the number.	
T1/DPN/S19	Printing patterns like this one and the ones in Exercises 4.18 and 4.19 is a good exercise for practicing *loop* control *statements*.	

T1/DPN/S20	The key is to understand the pattern and to describe it using *loop* control variables. Understand the pattern and to describe it using *loop* control variables.	
T1/DPN/S21	The last line in the outer *loop* (*line* 31), *System.out.println(),* does not have any argument in the *method*.	
T1/DPN/S22	This call moves the cursor to the next line.	
4.9	KEYWORDS BREAK AND CONTINUE	
T1/KBC/S1	Two *statements*, *break* and *continue*, can be used in *loop statements* to provide the *loop* with additional control. Use the two *statements*, *break* and *continue* in *loop statements* to provide the *loop* with additional control.	Dec/St/Co
T1/KBC/S2	*break* immediately ends the innermost *loop* that contains it.	
T1/KBC/S3	**It is generally used** || with an *if statement*. Use it with an *if statement.*	Dec/St/Co & Rel/Ob
T1/KBC/S4	*continue* only ends the current iteration.	
T1/KBC/S5	Program control goes to the end of the *loop* body.	
T1/KBC/S6	**This keyword is generally used** || with an *if statement*. Use this keyword with an *if statement.*	Dec/St/Co & Rel/Ob
T1/KBC/S7	You have already used the keyword *break* in a *switch statement*.	
T1/KBC/S8	You can also use *break* and *continue* in a *loop*. Use *break* and *continue* in a *loop*.	Dec/St/Co
T1/KBC/S9	*Listings 4.9* and *4.10* present two programs to demonstrate the effect of the *break* and *continue* keywords in a *loop*. See *Listings 4.9* and *4.10* that present two programs which demonstrate the effect of the *break* and *continue* keywords in a *loop*.	
T1/KBC/S10	The program in *Listing 4.9* adds the integers from *1 to 20* in this order to *sum* until *sum* is greater than or equal to *100.*	
T1/KBC/S11	Without the *if statement* (*line* 10), the program calculates the *sum* of the numbers from *1 to 20.*	
T1/KBC/S12	But with the *if statement*, the *loop* terminates when the *sum* becomes greater than or equal to *100.*	
T1/KBC/S13	The output of the program is shown in *Figure 4.11(a)*.	
T1/KBC/S14	If you changed the *if statement* as shown below, the output would resemble that in *Figure 4.11(b).*	
T1/KBC/S15	In this case, the *if* condition will never be true.	
T1/KBC/S16	Therefore, the *break statement* will never be executed.	
T1/KBC/S17	The program in *Listing 4.10* adds all the integers from *1 to 20* except *10 and 11* to *sum*.	
T1/KBC/S18	With the *if statement* in the program (*line* 9), the *continue statement* is executed when number becomes *10 or 11.*	
T1/KBC/S19	The *continue statement* ends the current iteration so that the rest of the *statement* in the *loop* body is not executed; therefore, number is not added to *sum* when it is 10 or 11.	
T1/KBC/S20	The output of the program is shown in *Figure 4.12(a)*. See the output of the program in *Figure 4.12(a)*.	Dec/St/Co
T1/KBC/S21	Without the *if statement* in the program, the output would look like *Figure 4.12(b).* Without the *if statement* in the program, see how the output would look like in *Figure 4.12(b)*.	
T1/KBC/S22	In this case, all of the numbers are added to *sum*, even when number is *10 or 11.*	

T1/KBC/S23	Therefore, the result is *210*, which is *21* more than it was with the *if statement*.	

Appendix 10B

Text 2 – Features of Metaphor of Mood and Metaphor Modality

Notes:
1. For every table entry with metaphorical clauses with Metaphor of Mood, the metaphorical sentence precedes the congruent sentence.
2. Mental projection clauses are in bold and separated with oblique lines (||).
3. Relational projection clauses are in bold and separated with oblique lines (||).

Keys:
1. Dec/St/Co denotes Declarative clauses realizing Statement and Command
2. Int/Qu/St denotes Interrogative clauses realizing Question and Statement
3. Int/Qu/Co denotes Interrogative clauses realizing Question and Command
4. Imp/Co/St denotes Imperative clauses realizing Command and Statement
5. Men/Prob denotes Mental Projection clause with Probability
6. Men/Ob denotes Mental Projection clause with Obligation
7. Rel/Prob denotes Relational projection clause with Probability

CHAPTER 9	FEELING A LITTLE *LOOP*Y	Code
TITLE	A *LOOP* FOR EVERY OCCASION (LEO)	
T2/LEO/S1	Have you ever been talking to someone and **it seems like** || he or she is saying the same thing over and over? There is probably a time when you talk to someone and it seems like he or she is saying the same thing over and over.	Int/Qu/St & Rel/Prob
T2/LEO/S2	**I mean,** || you keep listening, and they keep talking, and it all sounds the same. I mean, you keep listening, and they keep talking, doesn't it all sound the same?	Men/Ob
T2/LEO/S3	And they talk somemore and you listen somemore and **you wonder** || if it will ever end! And they talk somemore and you listen somemore and you wonder, will it ever end?	Men/Prob
T2/LEO/S4	Congratulations, **you just experienced** || a perfect example of a verbal *loop*! Congratulations, experience a perfect example of a verbal *loop!*	Dec/St/Co & Men/Prob
T2/LEO/S5	In *Java*, a *loop* is a programming construct that enables you to repeat a section of code over and over, much like my conversation example.	
T2/LEO/S6	*Loops* are very valuable in *Java* because they enable you to tightly control repetitive functions.	
T2/LEO/S7	Three type of *loops* are used in *Java*: *for loops*, *while loops* and *do loops*.	
TITLE	GETTING REDUNDANT WITH THE *FOR LOOP* (GRL)	

T2/GRL/S1	Let's pretend NASA used *Java* applets to control the launch of the space shuttle. Pretend NASA used *Java* applets to control the launch of the space shuttle.	Dec/St/Co
T2/GRL/S2	Any ideas on how controllers would initiate the launch sequence? Think of how controllers would initiate the launch sequence.	Int/Qu/Co
T2/GRL/S3	With *loops*!	
T2/GRL/S4	Counting down from ten to one is a piece of cake with a *loop*.	
T2/GRL/S5	Granted, without a *loop* it wouldn't be too tough either, but it would require some unnecessary code.	
T2/GRL/S6	Following is code to perform the launch sequence without the use of a *loop*. See the following for the code to perform the launch sequence without the use of a *loop*.	Dec/St/Co
T2/GRL/S7	And now the *loop* version: And now see the *loop* version:	Dec/St/Co
T2/GRL/S8	See what I mean about tightening up the code? This is what I mean about tightening up the code.	Int/Qu/St
T2/GRL/S9	**You probably wonder** || exactly how the *loop* code works. Think of how does the loop code work.	Dec/St/Co & Men/Prob
T2/GRL/S10	This code relies on a *for loop*, which is the most structured type of *loop* supported by *Java*.	
T2/GRL/S11	*For loops* repeat a section of code a fixed number of times.	
T2/GRL/S12	Following is the syntax for the *for loop*: See the following for the syntax for the *for loop.*	
T2/GRL/S13	The *for loop* repeats the *Statement* the number of time determined by the *InitializationExpression, LoopCondition* and *StepExpression:*	
T2/GRL/S14	The *InitializationExpression* is used to initialize a *loop* control variable. Use the *InitializationExpression* to initialize a *loop* control variable.	Dec/St/Co
T2/GRL/S15	The *LoopCondition* compares the *loop* control variable to some limit or value.	
T2/GRL/S16	The *StepExpression* specifies how the *loop* control variable should be modified before the next iteration of the *loop*.	
T2/GRL/S17	Let's take a look at the NASA launch sequence code again to make some sense of this stuff. Take a look at the NASA launch sequence code again to make some sense of this stuff.	Dec/St/Co
T2/GRL/S18	In this code the *InitializationExpression* is *int i310*, which is evaluated initially before the *loop* begins.	
T2/GRL/S19	This is the code you use to prime the *loop* and get it ready. Use this code to prime the *loop* and get it ready.	Dec/St/Co
T2/GRL/S20	The *LoopCondition* is *i>0*, which is a *Boolean* test that is performed before each iteration of the *loop*.	
T2/GRL/S21	If the *Boolean* test result is true, the *Statement* is executed, which in this case prints the current value of *i.*	
T2/GRL/S22	After each iteration the *StepExpression* is evaluated, which is *i--.*	
T2/GRL/S23	This serves to decrement *i* after each iteration, and ultimately proves the countdown.	
T2/GRL/S24	The *loop* continues to iterate and print numbers as *i* counts down to 0.	
T2/GRL/S25	After *i* reaches 0, the *LoopCondition* test fails (i>0), so the *loop* bails out without printing any more numbers.	
T2/GRL/S26	Whew, **that explanation seemed** || a little long-winded, and that's coming from the person that wrote it!	Men/Prob
T2/GRL/S27	Unfortunately, **it isn't always easy** || to verbalize the flow of program code.	Rel/Prob

T2/GRL/S28	**This is why it's easy** || to fall back on figures.	Rel/Prob
T2/GRL/S29	Just ask Ross Perot, who isn't a *Java* programmer but who nonetheless relied on diagrams and illustrations to help us grasp his big plans for the presidency. You can ask Ross Perot, who isn't a *Java* programmer but who nonetheless relied on diagrams and illustrations to help us grasp his big plans for the presidency.	Imp/Co/St
T2/GRL/S30	**You can feel safe and secure knowing that** || I'm not running *for* president or trying to help you visualize my answer to global trade. Be safe and secure knowing that I'm not running *for* president or trying to help you visualize my answer to global trade.	Dec/St/Co & Men/Prob
T2/GRL/S31	**I just want** || to help you learn how *loops* work!	Men/Ob
T2/GRL/S32	To help you visualize the *looping* process, take a look at the following figure. To help you visualize the *looping* process, you can take a look at the following figure.	Imp/Co/St
T2/GRL/S33	Notice in the figure that *Statement 1* and *Statement 2* will be repeatedly executed as long as the *loop* condition is true. You can notice in the figure that *Statement 1* and *Statement 2* will be repeatedly executed as long as the *loop* condition is true.	Imp/Co/St
T2/GRL/S34	When the *loop* condition goes false, the program falls out of the *loop* and executes *Statement 3.*	
T2/GRL/S35	The previous figure alludes to the fact that a *loop* can execute multiple *statements*.	
T2/GRL/S36	*Loops* can execute as many *statements* as they want, provided curly braces ({}) enclose the *statements.*	
T2/GRL/S37	If you recall, this grouping of *statements* is known as a *compound statement* and was used in the previous chapter when dealing with *if-else* branches. Recall that this grouping of *statements* is known as a *compound statement* and was used in the previous chapter when dealing with *if-else* branches.	Dec/St/Co
T2/GRL/S38	Following is an example of a *for loop* with a *compound statement:* See the following for an example of a *for loop* with a *compound statement:*	Dec/St/Co
T2/GRL/S39	This code calculates the squares of the numbers 1 through 10, stores them in an array, and prints each one.	
T2/GRL/S40	Notice that the *loop counter (i)* is used as the index *(i-1)* into the squares array. You can notice that the *loop counter (i)* is used as the index *(i-1)* into the squares array.	Imp/Co/St
T2/GRL/S41	**This is a very popular way** || to handle arrays.	Rel/Prob
T2/GRL/S42	**It is necessary** || to subtract 1 in this case because all *Java* array indexes start with 0, which means they are zero based. Subtract 1 in this case because all *Java* array indexes start with 0, which means they are zero based.	Dec/St/Co & Rel/Prob
T2/GRL/S43	**It might be worth nothing that** || although zero-based arrays were used in other programming languages in the 1980s and before, they have nothing to *do* with the 80s movie Less than Zero or the 80s hit song saved by Zero.	Rel/Ob
T2/GRL/S44	Rest assured I would be the first to tell you if they did! You can be rest assured I would be the first to tell you if they did!	Imp/Co/St
TITLE	LOOPING FOR JUST A LITTLE WHILE (LJLW)	
T2/LJLW/S1	Like the *for loop*, the *while loop* has a *loop* condition that controls the number of times a *loop* is repeated.	

T2/LJLW/S2	However, the *while loop* has no *initialization* or *step expression.*	
T2/LJLW/S3	A *for loop* is like one of those friends who tells you a story three or four times and then waits for a response, whereas a *while loop* is like one of those friends who continues to repeat himself as long as you continue to listen.	
T2/LJLW/S4	They're both annoying, but in different ways.	
T2/LJLW/S5	Not the *loops*, the people!	
T2/LJLW/S6	Following is the syntax *for* the *while loop*, which should make its usage a little more clear: See the following for the syntax for the *while loop*, which should make its usage a little more clear:	Dec/St/Co
T2/LJLW/S7	If the *Boolean LoopCondition* evaluates to true, the *Statement* is executed.	
T2/LJLW/S8	When the *Statement* finishes executing, the *LoopCondition* is tested again and the process repeats itself.	
T2/LJLW/S9	This continues until the *LoopCondition* evaluates to false, in which case the *loop* immediately bails out.	
T2/LJLW/S10	Because the *while loop* has no *step expression*, **it is important to make sure that** || the *Statement* somehow impacts the *LoopCondition*. Because the *while loop* has no *step expression*, make sure that the *Statement* somehow impacts the *LoopCondition*.	Dec/St/Co & Rel/Ob
T2/LJLW/S11	Otherwise, **it is possible** || for the *loop* to repeat infinitely, which is usually a bad thing.	Rel/Ob
T2/LJLW/S12	Following is a simple example of an infinite *while loop*: See the following for a simple example of an infinite *while loop*:	Dec/St/Co
T2/LJLW/S13	Because the *loop* condition in this example is permanently set to true, the *loop* will repeat infinitely, or at least until you manually terminate the programme.	
T2/LJLW/S14	Infinite *loops* are extremely dangerous because they can result in your computer overheating.	
T2/LJLW/S15	Just kidding!	
T2/LJLW/S16	Actually, infinite *loops* are useful in some situations.	
T2/LJLW/S17	They are never truly infinite because you can typically terminate one by shutting down the application or applet containing it.	
T2/LJLW/S18	**You can think of** || the *while loop* as a more general *for loop*. Think of the *while loop* as a more general *for loop*.	Dec/St/Co & Men/Ob
T2/LJLW/S19	To understand what I mean by this, check out the following code. To understand what I mean by this, you can check out the following code.	Imp/Co/St
T2/LJLW/S20	This is the NASA launch sequence implemented using a *while loop* instead of a *for loop*.	
T2/LJLW/S21	Because *while loops* don't have *initialization* expressions, the *initialization* of the *counter* variable *i* had to be performed before the *loop*.	
T2/LJLW/S22	Likewise, the *step expression i* had to be performed within the *Statement* part of the *loop*.	
T2/LJLW/S23	Regardless of the structural differences, this *while loop* is functionally equivalent to the *for loop* you saw earlier in the chapter.	
T2/LJLW/S24	If a *for loop* can do everything a *while loop* can and in a more organized way, then why do we need *while loops*? Even when a *for loop* can do everything a *while loop* can and in a more organized way, we still need *while loops.*	Int/Qu/St

T2/LJLW/S25	Because there is a time and a place for everything, and in many situations you have no need *for initialization* and *step expressions.*	
T2/LJLW/S26	A *for loop* is overkill in situations like this.	
T2/LJLW/S27	Even more importantly, a *while loop* is much more readable than a *for loop* when you have no need *for initialization* and *step expressions.*	
T2/LJLW/S28	Consider the following example: You can consider the following example:	Imp/Co/St
T2/LJLW/S29	This code demonstrates how a *while loop* could be used to ask a question and patiently wait for the correct answer. See this code that demonstrates how a *while loop* could be used to ask a question and patiently wait for the correct answer.	Dec/St/Co
T2/LJLW/S30	The *loop* repeats itself as long as the *Boolean* variable correct is false.	
T2/LJLW/S31	This results in the code repeating the question as many times as necessary until the user guesses the correct answer.	
T2/LJLW/S32	The details of the methods *askQuestion()* and *isCorrect()* aren't important for this example.	
T2/LJLW/S33	Just assume that they somehow present the user with a question, retrieve an answer, and then judge the correctness of the answer. You can assume that they somehow present the user with a question, retrieve an answer, and then judge the correctness of the answer.	Imp/Co/St
T2/LJLW/S34	**The main concern is that** || the *isCorrect () method* returns a *Boolean* value that indicates whether or not the answer is correct.	Rel/Prob
T2/LJLW/S35	In this example, **it is impossible** || to know how many times the user will miss the answer and need the question repeated.	Rel/Prob
T2/LJLW/S36	For this reason, the structured *step expression* of a *for loop* wouldn't be of much use.	
T2/LJLW/S37	While *loops* are perfect in situations where you don't know ahead of time how many times a *loop* needs to be repeated.	
T2/LJLW/S38	If you aren't completely satisfied with *while loops*, however, there is one other option. If you aren't completely satisfied with *while loops*, however, consider one other option.	Dec/St/Co
TITLE	TO *DO* , OR NOT TO *DO* (TDNTD)	
T2/TDNTD/S1	The *while loop* has a very close relative known as the *do loop*, or *do-while loop*, that is surprisingly similar to the *while loop*.	
T2/TDNTD/S2	Because you're becoming pretty *loop* savvy, I'll show you the syntax *for* the *do-while loop* first and see if you can figure out how it works. Because you're becoming pretty *loop* savvy, take note on the syntax *for* the *do-while loop* first and see if you can figure out how it works.	Dec/St/Co
T2/TDNTD/S3	Give up? If you can't figure it out, the answer is as follows: Give up now because the answer is as follows:	 Int/Qu/St Int/Qu/Co
T2/TDNTD/S4	The *do-while loop* is basically a *while loop* with the *LoopCondition* moved to the end.	

T2/TDNTD/S5	Why is this necessary? This is necessary.	Int/Qu/St
T2/TDNTD/S6	Because there are some situations where you would like the *Statement* to execute before evaluating the *LoopCondition*, instead of afterward.	
T2/TDNTD/S7	This also guarantees that the *Statement* is executed at least once, regardless of the *LoopCondition*.	
T2/TDNTD/S8	Let's take a look at the question and answer example implemented using a *do-while loop*: You can take a look at the question and answer example implemented using a *do-while loop*:	Imp/Co/St
T2/TDNTD/S9	The code really isn't much different than before, except that you no longer need to *initialise* the correct variable.	
T2/TDNTD/S10	**It is always initially set** || during the first pass through the *loop*.	Rel/Prob
T2/TDNTD/S11	Although both types of *loops* accomplish the goal of this example, the *do-while loop* is a better fit because of its structure more closely mimics the function of the code.	
T2/TDNTD/S12	What do I mean by this? Read on to know what I mean by this. The following describes what I mean by this.	 Int/Qu/Co Int/Qu/St
T2/TDNTD/S13	Well, if you "read" the code, it is saying "ask the question and if the answer is not correct, ask it again."	
T2/TDNTD/S14	**This makes more sense** || than if it read "if the answer is not correct, ask the question and then check the answer again."	Rel/Ob
T2/TDNTD/S15	Admittedly, this is a subtle difference, but a large part of successful programming is keeping things logical and straightforward.	
T2/TDNTD/S16	You won't always succeed because sometimes code gets complicated regardless of how you construct it, but using *loops* intelligently is a good start.	
TITLE	APPLET COUNTDOWN (AC)	
T2/AC/S1	Have you ever visited a Web page that directed you to another page, but informed you that if you waited a few second sit would automatically take you there? There is probably a time when you visit a Web page that directed you to another page, but informed you that if you waited a few second sit would automatically take you there.	Int/Qu/St
T2/AC/S2	I used to run across these pages and wonder how you could make a page wait a few seconds and then automatically navigate to a new page.	
T2/AC/S3	After I started programming in *Java*, **I realised** || what a trivial task this is.	Men/Ob
T2/AC/S4	In this section you use your knowledge of *loops* to build a "countdown" applet that counts down from ten to one and then navigates to a new Web page. In this section, use your knowledge of *loops* to build a "countdown" applet that counts down from ten to one and then navigates to a new Web page.	Dec/St/Co
T2/AC/S5	The following figure shows the *Countdown* applet in action. See the following figure that shows the *Countdown* applet in action.	Dec/St/Co
T2/AC/S6	When the applet finishes counting down, it navigates to the web page identified by the page applet parameter.	
T2/AC/S7	As an example, what web site could be better than NASA's to demonstrate how this applet works? As an example, think of what web site that could be better than NASA's to demonstrate how this applet works.	Dec/St/Co

T2/AC/S8	Following is NASA's Web site, to which the *Countdown* applet will take you after it finishes its countdown. See the following for NASA's Web site to which the *Countdown* applet will take you after it finishes its countdown.	Dec/St/Co
T2/AC/S9	To understand how the *Countdown* applet works, let's first take a look at the *Countdown. Html* Web page that contains the embedded applet. To understand how the *Countdown* applet works, you can first take a look at the *Countdown. Html* Web page that contains the embedded applet.	Imp/Co/St
T2/AC/S10	All this stuff should look pretty familiar to you by now.	
T2/AC/S11	The main thing on which **I want you** || to focus is the page parameter, which is defined as: Focus on the page parameter, which is defined as:	Dec/St/Co & Men/Ob
T2/AC/S12	Notice that the value of the page parameter is set to http://www.nasa.gov, which is the URL of NASA's Web site. You can notice that the value of the page parameter is set to http://www.nasa.gov, which is the URL of NASA's Web site.	Imp/Co/St
T2/AC/S13	Changing this value enables you to change the page that is loaded after the applet finishes counting down.	
T2/AC/S14	This page could have easily been set as a variable within the applet code, but a recompile would be required to change the page.	
T2/AC/S15	That is the beauty of applet parameters.	
T2/AC/S16	They enable you to customize the function of applets without doing any real programming! Customize the function of applets without doing any real programming!	Dec/St/Co
T2/AC/S17	Let's move on to the actual code required of the countdown applet. You can move on to the actual code required of the countdown applet.	Imp/Co/St
T2/AC/S18	Unfortunately, the *Countdown* applet requires some code that is a little beyond the lesson, so I don't expect all of this applet to make sense to you.	
T2/AC/S19	However, you can download the complete source code for the applet from the book's companion Web site, which was mentioned a little earlier in this section.	
T2/AC/S20	Also, the core mechanics of the applet are very straightforward and should be familiar to you from your recent study of *loops*.	
T2/AC/S21	Following is the *run() method* in the *Countdown* applet class, which forms the heart of the applet: See the following for the *run() method* in the *Countdown* applet class, which forms the heart of the applet:	Dec/St/Co
T2/AC/S22	Ouch, that looks a little messy!	
T2/AC/S23	Try not to get intimidated by any code that doesn't look familiar. You must try not to get intimidated by any code that doesn't look familiar.	Imp/Co/St
T2/AC/S24	Just concentrate on the *loop* code. You can just concentrate on the *loop* code.	Imp/Co/St
T2/AC/S25	As you can see, the *for loop* counts down from 10 to 1 just like the *Countdown* code you saw earlier in the chapter. See that the *for loop* counts down from 10 to 1 just like the *Countdown* code you saw earlier in the chapter.	Dec/St/Co
T2/AC/S26	The *Statement* part of this *for loop* is completely new territory, however.	

T2/AC/S27	The call to the *repaint () method* is necessary to update the applet's window with the new *countdown* number.	
T2/AC/S28	The call to the *thread.sleep() method* results in the applet waiting one second, which effectively pauses the *countdown* for one second between numbers.	
T2/AC/S29	When the *for loop* finishes, the code gets the page applet parameter and proceed to navigate to the Web page identified by it.	
T2/AC/S30	The code required to navigate to the Web page is probably pretty strange looking to you because it has to deal with *exceptions.*	
T2/AC/S31	*Exceptions* are errors caused by unforeseen problems such as your computer running out of memory, your modem coming unplugged, spilling coffee on your keyboard, hurling your monitor out the window, and so on.	
T2/AC/S32	I'll explain *exceptions* as you encounter them throughout the book. Be ready for explanation of *exceptions* as you encounter them throughout the book.	Dec/St/Co
T2/AC/S33	The complete source code for the *countdow*n applet is as follows. See the complete source code for the *countdow*n applet as follows.	Dec/St/Co
T2/AC/S34	Although this is a longer program than you are accustomed to seeing, a lot of it should look familiar to you.	
T2/AC/S35	For example, the *paint() method* code is very similar to the code used in the *DateTime* applet from Chapter 4, "constructing Applets of Your Own."	
T2/AC/S36	On the other hand, the *start() method* is entirely new and is related to the applet's use of *threads.*	
T2/AC/S37	You don't need to understand it fully at this point.	
TITLE	BREAKING AWAY (BA)	
T2/BA/S1	If you recall from the previous chapter, each case section of a *switch* branch ends with a *break statement*. Recall from the previous chapter, each case section of a *switch* branch ends with a *break statement*.	Dec/St/Co
T2/BA/S2	Following is an example to recap: See the following for an example to recap:	Dec/St/Co
T2/BA/S3	The purpose of the *break statement* in this example is to bail out of the *switch* branch so that no other code is executed.	
T2/BA/S4	The *break statement* serves a similar purpose in *loops*.	
T2/BA/S5	It breaks out of a *loop* regardless of the *loop* condition.	
T2/BA/S6	Following is an example of circumventing an infinite *loop* with a *break statement*: See the following for an example of circumventing an infinite *loop* with a *break statement*:	Dec/St/Co
T2/BA/S7	Without the assistance of the *break statement*, this *while loop* would continue forever thanks to the permanent *true loop* condition.	
T2/BA/S8	The *break statement* sidesteps this problem by breaking out of the *loop* after one hundred iterations (0-99).	
T2/BA/S9	Of course, **it is rare that** || you would purposely create an infinite *loop* and then use a *break statement* to bail out of it.	Rel/Prob
T2/BA/S10	However, the *break statement* can be very useful in some tricky *loops* when you need to exit at an otherwise inconvenient time.	
T2/BA/S11	A close relative of the *break statement* is the *continue statement*, which is used to skip to the next iteration of a *loop*.	
T2/BA/S13	The following example shows how a *continue statement* can be used to *print* only the even numbers between	Dec/St/Co

	1 and 100: See the following example that shows how a *continue statement* can be used to *print* only the even numbers between 1 and 100:	
T2/BA/S14	Having trouble seeing how this one works? You might have trouble seeing how this one works.	Int/Qu/St
T2/BA/S15	Think back to the *modulus operator* (%), which returns the remainder of a division. You can think back to the *modulus operator* (%), which returns the remainder of a division.	Imp/Co/St
T2/BA/S16	Now consider what the remainder of a division by 2 yields for even and odd numbers. Now, you can consider what the remainder of a division by 2 yields for even and odd numbers.	Imp/Co/St
T2/BA/S17	Aha!	
T2/BA/S18	Even numbers divided by 2 always yield a remainder of 0, and odd numbers always leave a remainder of 1!	
T2/BA/S19	The example code exploits this characteristic of even and odd numbers to skip to the next iteration bypasses the *println()* call, which prevents odd numbers from being printed. See that the example code exploits this characteristic of even and odd numbers to skip to the next iteration bypasses the *println()* call, which prevents odd numbers from being printed.	Dec/St/Co
T2/BA/S20	Pretty tricky!	
TITLE	THE LEAST YOU NEED TO KNOW (LNK)	
T2/LNK/S1	Computers are often called upon to perform tasks we humans find to be utterly redundant.	
T2/LNK/S2	As dull as some humans can be, **I guarantee you** || computers are much duller when it comes to repeating the same thing over and over.	Men/Ob
T2/LNK/S3	*Java* enables you to build programs that repeat themselves through the use of *loops*.	
T2/LNK/S4	The different types of *loops* basically perform the same function.	
T2/LNK/S5	They repeat a section of code over and over.	
T2/LNK/S6	Let's go over the main points you learned about *loops* in this chapter. You can go over the main points you learned about *loops* in this chapter.	Imp/Co/St
T2/LNK/S7	*Loops* can execute as many *statements* as you want them to, provided the *statements* are grouped together as a single *compound statement* enclosed by curly braces ({}).	
T2/LNK/S8	A *for loop* is used to repeat a section of code a given number of iterations. Use a *for loop* to repeat a section of code a given number of iterations.	Dec/St/Co
T2/LNK/S9	A *while loop* is a more general *for loop*.	
T2/LNK/S10	A *do while* is a *while loop* with the *loop* condition moved to the end.	
T2/LNK/S11	The *break statement* is used to break out of a *loop* regardless of the *loop* condition. Use the *break statement to break out of a loop* regardless of the *loop* condition.	Dec/St/Co
T2/LNK/S12	The *continue statement* is used to skip to the next iteration of a *loop*. Use the *continue statement* to skip to the next iteration of a *loop*.	Dec/St/Co