# Chapter 4

# Loops

## 4.1 Introduction

- Loops are structures that control **repeated** executions of a block of statements.
- Java provides a powerful control structure called a **loop**, which controls how many times an operation or a sequence of operation is performed in succession.
- Java provides three types of loop statements **while** loops, **do-while** loops, and **for** loops.

## 4.2 The while Loop

- The syntax for the while loop is as follows:

```
while (loop-continuation-condition) {
  // loop-body
  Statement(s);
}
```

- The braces enclosing a *while* loop or any other loop can be omitted only if the loop body contains one or no statement. The *while* loop flowchart is in Figure (a).
- The *loop-continuation-condition*, a Boolean expression, must appear inside the parentheses. It is always evaluated **before** the loop body is executed.
- If its evaluation is *true*, the loop body is executed; if its evaluation is false, the entire loop terminates, and the program control turns to the statement that follows the *while* loop.

- For example, the following *while* loop prints `Welcome to Java!` **100** times.

```
int count = 0;
while (count < 100) {
  System.out.println("Welcome to Java!");
  count++;
}
```

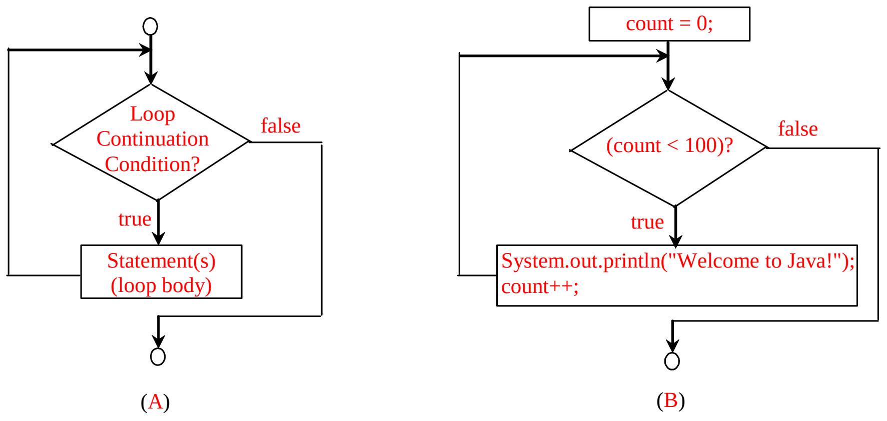


FIGURE 4.1 The while loop repeatedly executes the statements in the loop body when the loop-continuation-condition evaluates to true.

## Caution

- Make sure that the *loop-continuation-condition* **eventually** becomes false so that the program will terminate.
- A common programming error involves **infinite** loops.

## 4.2.1 Problem: Guessing Numbers (Page 118)

- Write a program that randomly generates an integer **between 0 and 100, inclusive**. The program prompts the user to enter a number continuously until the number matches the randomly generated number. For each user input, the program tells the user whether the input is too low or too high, so the user can choose the next input intelligently.

- LISTING 4.2 GuessNumber.java

```
import java.util.Scanner;

public class GuessNumber {
  public static void main(String[] args) {
    // Generate a random number to be guessed
    int number = (int)(Math.random() * 101);

    Scanner input = new Scanner(System.in);
    System.out.println("Guess a magic number between 0 and 100");

    int guess = -1;
    while (guess != number) {
     // Prompt the user to guess the number
      System.out.print("\nEnter your guess: ");
      guess = input.nextInt();

      if (guess == number)
        System.out.println("Yes, the number is " + number);
      else if (guess > number)
        System.out.println("Your guess is too high");
      else
        System.out.println("Your guess is too low");
    } // End of loop
  }
}
```

```
Guess a magic number between 0 and 100

Enter your guess: 50
Your guess is too high

Enter your guess: 25
Your guess is too high

Enter your guess: 12
Your guess is too high

Enter your guess: 6
Your guess is too low

Enter your guess: 9
Yes, the number is 9
```

## 4.2.3 Example: An Advanced Math Learning Tool (Page 121)

- The Math subtraction learning tool program generates just one question for each run. You can use a **loop** to generate questions repeatedly. This example gives a program that generates five questions and reports the number of the correct answers after a student answers all **five** questions.

- LISTING 4.3 SubtractionQuizLoop.java

```
import java.util.Scanner;

public class SubtractionQuizLoop {
  public static void main(String[] args) {
    final int NUMBER_OF_QUESTIONS = 5; // Number of questions
    int correctCount = 0; // Count the number of correct answers
    int count = 0; // Count the number of questions
    long startTime = System.currentTimeMillis();
    String output = ""; // output string is initially empty
    Scanner input = new Scanner(System.in);

    while (count < NUMBER_OF_QUESTIONS) {
      // 1. Generate two random single-digit integers
      int number1 = (int)(Math.random() * 10);
     int number2 = (int)(Math.random() * 10);

      // 2. If number1 < number2, swap number1 with number2
      if (number1 < number2) {
        int temp = number1;
        number1 = number2;
        number2 = temp;
      }

      // 3. Prompt the student to answer “What is number1 – number2?”
      System.out.print(
        "What is " + number1 + " - " + number2 + "? ");
      int answer = input.nextInt();

      // 4. Grade the answer and display the result
      if (number1 - number2 == answer) {
       System.out.println("You are correct!");
        correctCount++;
      }
      else
        System.out.println("Your answer is wrong.\n" + number1
          + " - " + number2 + " should be " + (number1 - number2));

      // Increase the count
      count++;

      output += "\n" + number1 + "-" + number2 + "=" + answer +
        ((number1 - number2 == answer) ? " correct" : " wrong");
    }
    long endTime = System.currentTimeMillis();
    long testTime = endTime - startTime;

    System.out.println("Correct count is " + correctCount +
      "\nTest time is " + testTime / 1000 + " seconds\n" + output);
  }
}
```

```
What is 9 - 2? 7
Your answer is correct!

What is 3 - 0? 3
Your answer is correct!

What is 3 - 2? 1
Your answer is correct!

What is 7 - 4? 4
Your answer is wrong.
7 - 4 should be 3

What is 7 - 5? 4
Your answer is wrong.
7 - 5 should be 2

Correct count is 3
Test time is 1021 seconds

9-2=7 correct
3-0=3 correct
3-2=1 correct
7-4=4 wrong
7-5=4 wrong
```

## 4.2.4 Controlling a Loop with a Sentinel Value

- Often the number of times a loop is executed is not predetermined. You may use an input value to signify the **end** of the loop. Such a value is known as a sentinel value.
- Write a program that reads and calculates the sum of an unspecified number of integers. The input 0 signifies the end of the input.

- LISTING 4.4 SentinelValue.java (**Page 123**)

```
import java.util.Scanner;

public class SentinelValue {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Read an initial data
    System.out.print(
      "Enter an int value (the program exits if the input is 0): ");
    int data = input.nextInt();

    // Keep reading data until the input is 0
    int sum = 0;
    while (data != 0) {
     sum += data;

      // Read the next data
      System.out.print(
        "Enter an int value (the program exits if the input is 0): ");
      data = input.nextInt();
    }

    System.out.println("The sum is " + sum);
  }
}
```

```
Enter an int value (the program exits if the input is 0): 2
Enter an int value (the program exits if the input is 0): 3
Enter an int value (the program exits if the input is 0): 4
Enter an int value (the program exits if the input is 0): 0
The sum is 9
```

- If data is not 0, it is added to the sum and the next input data are read. If data is **0**, the loop body is not executed and the while loop terminates.
- If the first input read is 0, the loop body **never** executes, and the resulting *sum* is 0.
- The **do-while** loop executes the loop body first, and then checks the loop-continuation condition to determine whether to continue or terminate the loop.

## Caution

- **Don't use floating-point values for equality** checking in a loop control. Since floating-point values are **approximations** for some values, using them could result in imprecise counter values and inaccurate results. Consider the following code for computing 1 + 0.9 + 0.8 + ... + 0.1:

```
double item = 1; double sum = 0;
while (item != 0) { // No guarantee item will be 0
  sum += item;
  item -= 0.1;
}
System.out.println(sum);
```

- Variable item starts with 1 and is reduced by 0.1 every time the loop body is executed. The loop should terminate when item becomes 0. However, there is no guarantee that item will be **exactly 0**, because the floating-point arithmetic is **approximated**. This loop seems OK on the surface, but it is actually an **infinite** loop.

### 4.2.5 Input and Output Redirections

- If you have a large number of data to enter, it would be cumbersome to type from the key board. You may store the data separated by whitespaces in a text file, say input.txt, and run the program using the following command:

```
java SentinelValue < input.txt
```

- This command is called **input redirection**. The program takes the input from the file input.txt rather thatn having the user to type the data from the keyboard at runtime.

- There is **output redirection** which sends the output to a file rather than displaying it on the console. The command for output redirection is:

```
java ClassName > output.txt
```

- Input and output redirection can be used in the same command. For example, the following command gets input from input.txt and sends output to output.txt:

```
java SentinelValue < input.txt > output.txt
```

## 4.3 The do-while Loop

- The *do-while* is a variation of the *while-loop*. Its syntax is shown below.

```
do {
   // Loop body
   Statement(s);
} while (continue-condition);          // Do not forget ";"
```

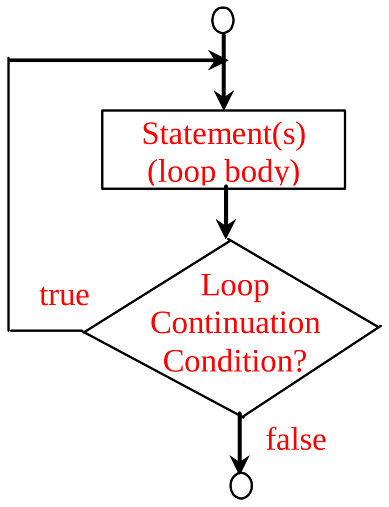


FIGURE 4.2 The do-while loop executes the loop body first, then checks the loop-continuation-condition to determine whether to continue or terminate the loop.

- The loop body is **executed first**. Then the *loop-continuation-condition* is evaluated.
- If the evaluation is *true*, the loop body is executed **again**; if it is *false*, the *do-while* loop terminates.
- 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.
- The *while* loop and the *do-while* loop have equal expressive power.
- Sometimes one is a more convenient choice than the other.
- **Tip:** Use the `do-while` loop if you have statements inside the loop that must be executed **at least once**.

- You can rewrite the TestWhile program shown previously as follows:

- LIST 4.5 TestDoWhile.java (**Page 125)**

```java
import java.util.Scanner;

public class TestDoWhile {
  /** Main method */
  public static void main(String[] args) {
    int data;
    int sum = 0;

    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Keep reading data until the input is 0
    do {
      // Read the next data
      System.out.print(
        "Enter an int value (the program exits if the input is 0): ");
      data = input.nextInt();

     sum += data;
    } while (data != 0);

    System.out.println("The sum is " + sum);
  }
}
```

```
Enter an int value (the program exits if the input is 0): 2
Enter an int value (the program exits if the input is 0): 3
Enter an int value (the program exits if the input is 0): 4
Enter an int value (the program exits if the input is 0): 0
The sum is 9
```

## 4.4 The for Loop

- The syntax of a for loop is as shown below.

```
for (initial-action; loop-continuation-condition;
      action-after-each-iteration) {
   //loop body;
   Statement(s);
}
```

- 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 the loop body, enclosed inside braces.
- *initial*-action, *loop-continuation-condition,* and *action-after-each-iteration* are separated by **semicolons**;
- A *for* loop generally uses a variable to control how many times the loop body is executed and when the loop terminates.
- This variable is referred to as a ***control variable***. 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.
- Example: The following for loop prints Welcome to Java! 100 times.

```
int i;
for (i = 0; i < 100; i++) {
  System.out.println("Welcome to Java! ”);
}
```

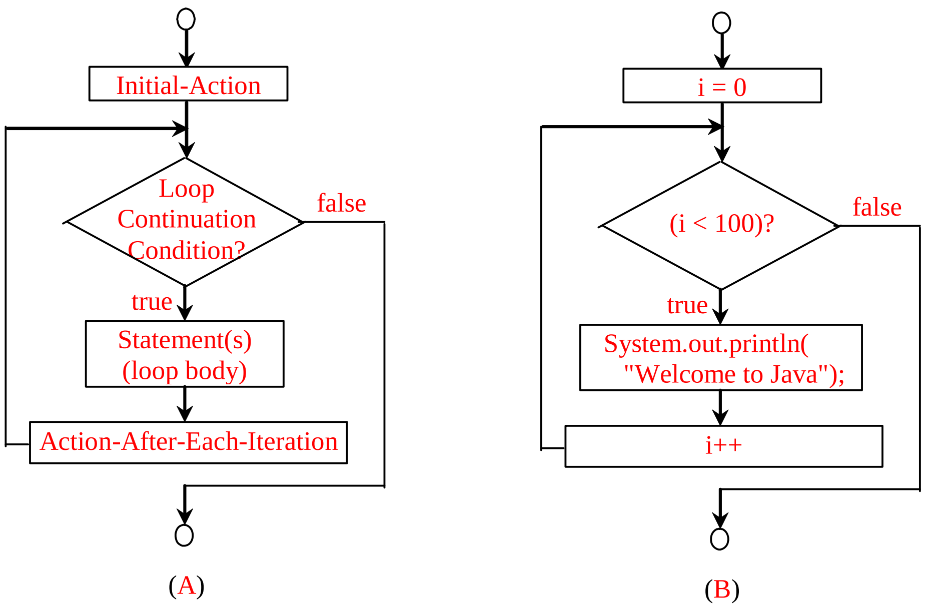

Initial-Action
Loop Continuation Condition?
false
true
Statement(s) (loop body)
Action-After-Each-Iteration
(A)
i = 0
(i < 100)?
false
true
System.out.println( "Welcome to Java");
i++
(B)

FIGURE 4.3 A for loop performs an initial action one, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the loop-continuation-condition evaluates as true

- The `for` loop initializes `i` to `0`, then repeatedly executes the `println` and evaluates `i++` if `i` is less than `100`.
- The *initial-action*, `i = 0`, initializes the control variable, `i`.
- The *loop-continuation-condition*, `i < 100`, is a Boolean expression.
- The expression is evaluated at the beginning of each iteration.
- If the condition is *true*, execute the loop body. If it is *false*, the loop terminates and the program control turns to the line following the loop.
- The *action-after-each-iteration*, `i++`, is a statement that adjusts the control variable.
- This statement is executed after each iteration. It increments the control variable.
- Eventually, the value of the control variable forces the *loop-continuation-condition* to become *false*.
- The loop control variable can be declared and initialized in the `for` loop as follows:

```
for (int i = 0; i < 100; i++) {
  System.out.println("Welcome to Java");
}
```

## Note

- The initial-action in a `for` loop can be a list of zero or more comma-separated variable declaration statements or assignment expressions.

```
for (int i = 0, j = 0; (i + j < 10); i++, j++) {
        // Do something
}
```

- The action-after-each-iteration in a `for` loop can be a list of zero or more comma-separated statements. The following is correct but not a good example, because it makes the code hard to read.

```
for (int i = 1; i < 100; System.out.println(i), i++);
```

## Note

- If the loop-continuation-condition in a `for` loop is omitted, it is implicitly **true**. Thus the statement given below in (A), which is an infinite loop, is correct. Nevertheless, I recommend that you use the equivalent loop in (B) to avoid confusion:

```
for ( ; ; ) {
  // Do something
}
```

(a)

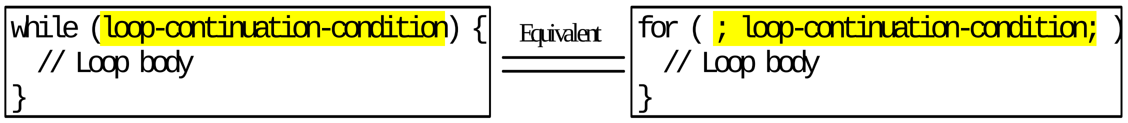
Equivalent

```
while (true) {
  // Do something
}
```

(b)

## 4.5 Which Loop to Use?

- The three forms of loop statements, ***while*, *do*, and *for***, are expressively equivalent; that is, you can write a loop in any of these three forms.
- For example, a `while` loop in (a) in the following figure can always be converted into the following `for` loop in (b):



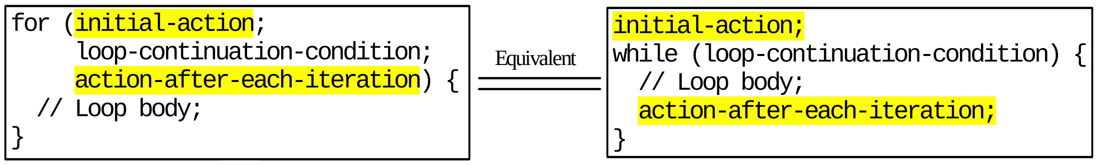
```
while (loop-continuation-condition) {
  // Loop body
}
```

(a)

```
for ( ; loop-continuation-condition; )
  // Loop body
}
```

(b)

- A for loop in (a) in the following figure can generally be converted into the following while loop in (b) except in certain special cases.



```
for (initial-action;
     loop-continuation-condition;
     action-after-each-iteration) {
  // Loop body;
}
```

(a)

```
initial-action;
while (loop-continuation-condition) {
  // Loop body;
  action-after-each-iteration;
}
```

(b)

## Recommendations

- The author recommends that you use the one that is most intuitive and comfortable for you.
- 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 100 times.
- 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.
- A `do-while` loop can be used to replace a `while` loop if the loop body has to be executed **before** testing the continuation condition.

## Caution

- Adding a semicolon at the end of the `for` clause before the loop body is a common mistake, as shown below:

```
for (int i = 0; i < 10; i++);          // Logic Error (‘;’)
{
  System.out.println("i is " + i);
}
```

- Similarly, the following loop is also wrong:

```
   int i=0;
   while (i<10);                       // Logic Error (‘;’)
   {
     System.out.println("i is " + i);
     i++;
   }
```

- In the case of the *do* loop, the following **semicolon is needed** to end the loop.

```
   int i=0;
   do {
     System.out.println("i is " + i);
     i++;
   } while (i<10);               // Correct, The semicolon is needed
```

## 4.6 Nested Loops

- Nested loops consist of an outer loop and one or more inner loops. Each time the outer loop is repeated, the inner loops are **reentered**, and **all** the required iterations are performed.

- Problem: Write a program that uses **nested** for loops to print a multiplication table.

- LISTING 4.6 MultiplicationTable.java (Page 129)

```java
public class MultiplicationTable {
  /** Main method */
  public static void main(String[] args) {
    // Display the table heading
    System.out.println("           Multiplication Table");

    // Display the number title
    System.out.print("    ");
    for (int j = 1; j <= 9; j++)
      System.out.print("   " + j);

    System.out.println("\n-----------------------------------------");

    // Print table body
    for (int i = 1; i <= 9; i++) {
      System.out.print(i + " | ");
      for (int j = 1; j <= 9; j++) {
        // Display the product and align properly
        System.out.printf("%4d", i * j);
      }
      System.out.println();
    }
  }
}
```

```
           Multiplication Table
       1   2   3   4   5   6   7   8   9
-----------------------------------------
1 |    1   2   3   4   5   6   7   8   9
2 |    2   4   6   8  10  12  14  16  18
3 |    3   6   9  12  15  18  21  24  27
4 |    4   8  12  16  20  24  28  32  36
5 |    5  10  15  20  25  30  35  40  45
6 |    6  12  18  24  30  36  42  48  54
7 |    7  14  21  28  35  42  49  56  63
8 |    8  16  24  32  40  48  56  64  72
9 |    9  18  27  36  45  54  63  72  81
```

## 4.7 Minimizing Numerical Errors

- Numeric errors involving floating-point numbers are inevitable.

- Write a program that sums a series that starts with 0.01 and ends with 1.0. The numbers in the series will increment by 0.01, as follows 0.01 + 0.02 + 0.03 and so on.
- LISTING 4.7 TestSum.java (**Page 130)**

```
public class TestSum {
  public static void main(String[] args) {
    // Initialize sum
    float sum = 0;

    // Add 0.01, 0.02, ..., 0.99, 1 to sum
    for (float i = 0.01f; i <= 1.0f; i = i + 0.01f)
      sum += i;

    // Display result
    System.out.println("The sum is " + sum);
  }
}
```

```
The sum is 50.499985
```

  - The for loop repeatedly adds the control variable i to the sum. This variable, which begins with 0.01, is incremented by 0.01 after each iteration. The loop terminates when i exceeds 1.0.
  - The exact sum should be **50.50**, but the answer is **50.499985**. 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 number exactly.

- If you change float in the program to **double** as follows, you should see a slight improvement in precision because a double variable takes 64 bits, whereas a float variable takes 32 bits.

```
public class TestSum {
  public static void main(String[] args) {
    // Initialize sum
    double sum = 0;

    // Add 0.01, 0.02, ..., 0.99, 1 to sum
    for (double i = 0.01; i <= 1.0; i = i + 0.01)
      sum += i;

    // Display result
    System.out.println("The sum is " + sum);
  }
}
```

```
The sum is 49.50000000000003
```

- To fix the problem: Using an integer count to ensure that all the numbers are processed.

```
public class TestSum {
   public static void main(String[] args) {
      // Initialize sum
      double sum = 0;
      double currentValue = 0.01;

      // Add 0.01, 0.02, ..., 0.99, 1 to sum
      for (int count = 0; count < 100; count++) {
         sum += currentValue;
         currentValue += 0.01;
      }

      // Display result
      System.out.println("The sum is " + sum);
  }
}
```

```
The sum is 50.50000000000003
```

## 4.8 Case Studies

- Control statements are fundamental in programming.
- The ability to write control statement is essential in learning Java programming.
- *If you can write programs using loops, you know how to program!*

### 4.8.1 Problem: Finding the Greatest Common Divisor (Page 131)

- Problem: Write a program that prompts the user to enter two positive integers and finds their **greatest common divisor**.
- LISTING 4.8 GreatestCommonDivisor.java (**Page 132**)
- Solution: Suppose you enter two integers 4 and 2, their greatest common divisor is 2. Suppose you enter two integers 16 and 24, their greatest common divisor is 8. So, how do you find the greatest common divisor? Let the two input integers be n1 and n2. You know number 1 is a common divisor, but it may not be the greatest commons divisor. 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.

```java
import java.util.Scanner;

public class GreatestCommonDivisor {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter first integer: ");
    int n1 = input.nextInt();
    System.out.print("Enter second integer: ");
    int n2 = input.nextInt();

    int gcd = 1;
    int k = 2;
    while (k <= n1 && k <= n2) {
      if (n1 % k == 0 && n2 % k == 0)
        gcd = k;
      k++;
    }

    System.out.println("The greatest common divisor for " + n1 +
      " and " + n2 + " is " + gcd);
  }
}
```

```
Enter first integer: 125
Enter second integer: 2525
The greatest common divisor for 125 and 2525 is 25
```

## 4.8.2 Problem: Predicating the Future Tuition (Page 133)

- Problem: Suppose that the tuition for a university is $10,000 this year and tuition increases **7%** every year. In how many years will the tuition be **doubled**?

```
double tuition = 10000;   int year = 1          // Year 1
tuition = tuition * 1.07; year++;               // Year 2
tuition = tuition * 1.07; year++;               // Year 3
tuition = tuition * 1.07; year++;               // Year 4
...
```

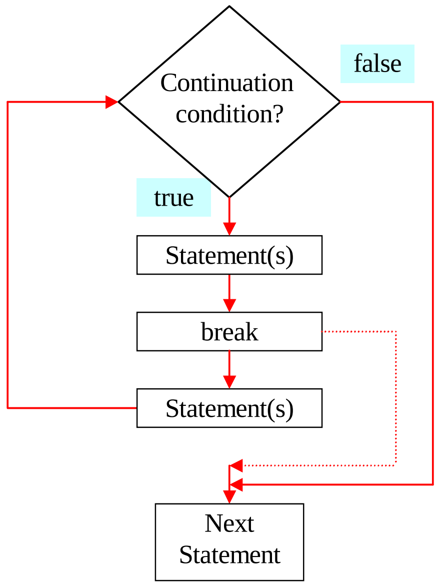
- LISTING 4.9 FutureTuition.java (**Page 133**)

```
public class FutureTuition {
  public static void main(String[] args) {
    double tuition = 10000;   // Year 1
    int year = 1;
    while (tuition < 20000) {
      tuition = tuition * 1.07;
      year++;
    }

    System.out.println("Tuition will be doubled in "
      + year + " years");
  }
}
```

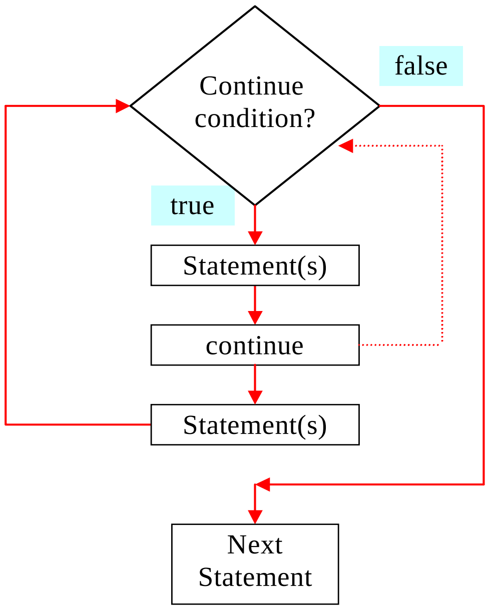
```
Tuition will be doubled in 12 years
```

## 4.9 Keywords break and continue

- The **`break`** control **immediately ends the innermost loop** that contains it. It is generally used with an `if` statement.
- The **`continue`** control **only ends the current iteration**. Program control goes to the end of the loop body. This keyword is generally used with an `if` statement.
- The **break** statement forces its containing loop to exit.



- The **continue** statement forces the current iteration of the loop to end.



## Demonstrating a break Statement

- LISTING 4.11 TestBreak.java (**Page 135)**
  - This program adds the integers from 1 to 20 in this order to sum until sum is greater than or equal to 100.

```
Output:
      The number is 14
      The sum is 105
```

  - Without the if statement, the program calculates the sum of the numbers from 1 to 20.

```
Output:
      The number is 20
      The sum is 210
```

```java
public class TestBreak {
  public static void main(String[] args) {
    int sum = 0;
    int number = 0;

    while (number < 20) {
      number++;
      sum += number;
      if (sum >= 100) break;
    }

    System.out.println("The number is " + number);
    System.out.println("The sum is " + sum);
  }
}
```

```
The number is 14
The sum is 105
```

## Demonstrating a continue Statement

- LISTING 4.12 TestContinue.java (**Page 136)**
    - This program adds all the integers from 1 to 20 except 10 and 11 to sum.

      ```
      Output:
            The sum is 189
      ```

    - Without the if statement in the program, all of the numbers are added to sum, even when number is 10 or 11.

      ```
      Output:
            The sum is 210
      ```

```
public class TestContinue {
  public static void main(String[] args) {
    int sum = 0;
    int number = 0;

    while (number < 20) {
      number++;
      if (number == 10 || number == 11) continue;
      sum += number;
    }

    System.out.println("The sum is " + sum);
  }
}
```

```
The sum is 189
```

## Statement Labels and Breaking with Labels (Optional)

- Every Statement in Java can have an optional label as an identifier. Labels are often used with **break** and **continue** statements.
- You can use a *break* statement with a label to break out of the labeled loop, and a *continue* statement with a label to break out of the current iteration of the labeled loop.
- The break statement given below, for example, breaks out of the outer loop `if (i * j) > 50` and transfers control to the statement immediately following the outer loop.

```
outer:
    for (int i = 1; i < 10; i++) {
    inner:
       for (int j = 1; j < 10; j++) {
           if (i * j > 50)
             break outer;
           System.out.println(i * j);
       }
    }
```

- If you replace *break outer* with *break* in the preceding statement, the *break* statement would break out of the inner loop and continue to stay inside the outer loop.

- The following *continue* statement breaks out of the inner loop if (i * j > 50) and starts a new iteration of the outer loop if i < 10 is true after i us incremented by 1:

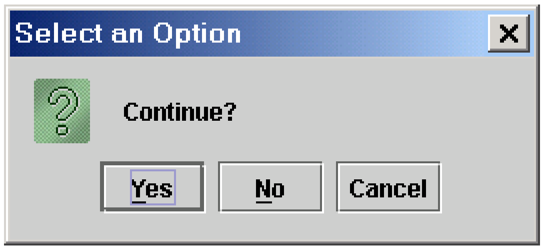
```
outer:
    for (int i = 1; i < 10; i++) {
    inner:
       for (int j = 1; j < 10; j++) {
           if (i * j > 50)
             continue outer;
           System.out.println(i * j);
       }
     }
```

- If you replace *continue outer* with *continue* in the preceding statement, the *continue* statement would break out of the current iteration of the inner loop `if (i * j > 50)` and continue the next iteration of the inner loop if j < 10 is true after j is incremented by 1.

## 4.10 (GUI) Controlling a Loop with a Confirmation Dialog

- A sentinel-controlled loop can be implemented using a **confirmation dialog**. The answers Yes or No to continue or terminate the loop.
- For example, the following loop continues to execute until the user clicks the No or Cancel button.

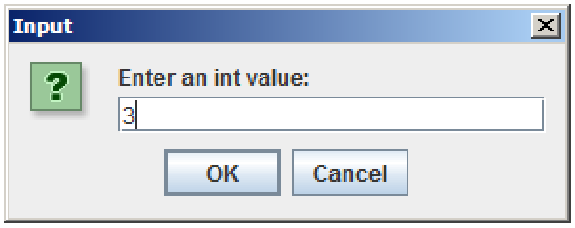
```
int option = 0;
while (option == JOptionPane.YES_OPTION) {
   System.out.println("continue loop");
   option = JOptionPane.showConfirmDialog(null, "Continue?");
}
```



- The value is
  - o JOptionPane.YES_OPTION (0) for Yes button,
  - o JOptionPane.NO_OPTION (1) for the No button, and
  - o JOptionPane.CANCEL_OPTION (2) for Cancel button.

- LISTING 4.15 SentinelValueUsingConfirmationDialog.java (**Page 136)**

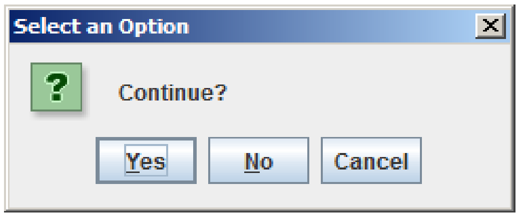
```java
import javax.swing.JOptionPane;

public class SentinelValueUsingConfirmationDialog {
  public static void main(String[] args) {
    int sum = 0;

    // Keep reading data until the user answers No
    int option = 0;
    while (option == JOptionPane.YES_OPTION) {
      // Read the next data
      String dataString = JOptionPane.showInputDialog(
        "Enter an int value: ");
      int data = Integer.parseInt(dataString);

      sum += data;

      option = JOptionPane.showConfirmDialog(null, "Continue?");
    }

    JOptionPane.showMessageDialog(null, "The sum is " + sum);
  }
}
```


Input

Enter an int value:

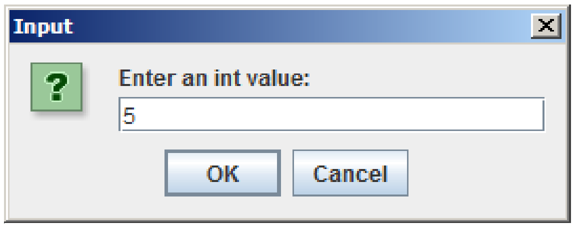
3

(a)


Select an Option

Continue?

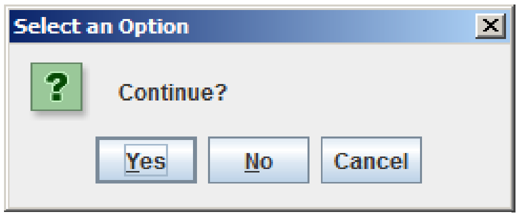
(b)


Input

Enter an int value:

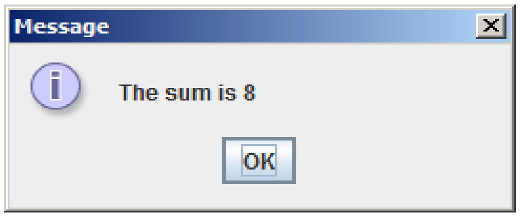
5

(c)


Select an Option

Continue?

(d)


Message

The sum is 8

(e)

FIGURE 4.4 The user enters 3 in (a), clicks Yes in (b), enters 5 in (c), clicks No in (d), and the result is shown in (e).