# Chapter 5

Loops



## Objectives

- To write programs for executing statements repeatedly using a **while** loop
- To control a loop with a sentinel value
- To write loops using **do-while** statements
- To write loops using **for** statements
- To discover the similarities and differences of three types of loop statements
- To write nested loops
- To implement program control with **break** and **continue**
- To obtain input from a file rather than typing from the keyboard

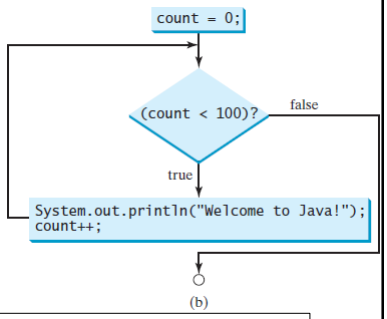
# Loops: an introduction

- *A loop tells a program to execute statements **repeatedly**.*
- Using a loop statement, you tell the computer to display a string a hundred times without having to code the print statement a hundred times, as follows:

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

loop-continuation-condition

loop body




- The loop checks whether **count < 100** is **true**. If so, it executes the loop body to display the message **Welcome to Java!** and increments **count** by **1**.
- It repeatedly executes the loop body until **count < 100** becomes **false**.
- When **count < 100** is **false** (i.e., when **count** reaches **100**), the loop terminates and the next statement after the loop statement is executed.
- In this example, you know exactly how many times the loop body needs to be executed because the control variable **count** is used to count the number of executions. This type of loop is known as a *counter-controlled loop.*



# Infinite loop: a loop that never ends

- It is important to have a statement in the loop body that will finally change the value of the loop-continuation-condition to false, so as to end the loop.

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

loop-continuation-condition

loop body

Try to trace the program in debugger of Eclipse.

- Imagine what will happen if the statement “count++;” is missing in the above example. The loop will run forever (an infinite loop that never ends).

# Loops: an introduction (cont'd)

- *Loops* control repeated executions of a block of statements.
- Java provides three types of loop statements:
  - **while** loops,
  - **do**-**while** loops, and
  - **for** loops





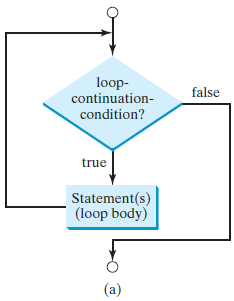
# The while loop

- *A* **while** *loop executes statements repeatedly while the condition is true.*

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

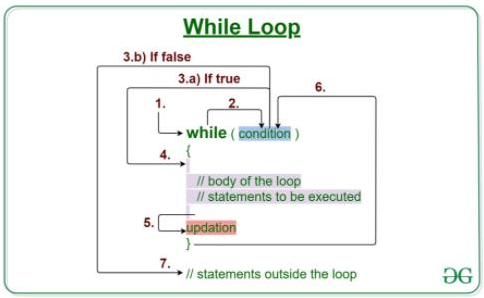
- The **loop-continuation-condition** must always appear inside the parentheses.
- The braces enclosing the loop body can be omitted only if the loop body contains one or no statement.

- The part of the loop that contains the statements to be repeated is called the *loop body.*
- A one-time execution of a loop body is referred to as an *iteration (or repetition) of the loop.*
- Each loop contains a *loop-continuation-condition,* a Boolean (true / false) expression that controls the execution of the body.
  - It is evaluated each time to decide if the loop body is executed.
  - If its evaluation is **true**, the loop body is executed; if its evaluation is **false**, the entire loop ends and the program control turns to the statement that follows the **while** loop.



- **while** loops
- **do-while** loops
- **for** loops

# while loop




Reference: https://www.geeksforgeeks.org/java-while-loop-with-examples/?ref=lbp

- Write a while loop that prints 0 to 100.



- **while** loops
- **do-while** loops
- **for** loops

# The while loop: an example

- Recall that Listing 3.1, AdditionQuiz.java, gives a program that prompts the user to enter an answer for a question on addition of two single digits.
- Using a loop, rewrite the program to let the user repeatedly enter a new answer until it is correct.

**LISTING 3.1** AdditionQuiz.java

```
1  import java.util.Scanner;
2
3  public class AdditionQuiz {
4    public static void main(String[] args) {
5      int number1 = (int)(System.currentTimeMillis() % 10);
6      int number2 = (int)(System.currentTimeMillis() / 7 % 10);
7
8      // Create a Scanner
9      Scanner input = new Scanner(System.in);
10
11     System.out.print(
12       "What is " + number1 + " + " + number2 + "? ");
13
14     int answer = input.nextInt();
15
16     System.out.println(
17       number1 + " + " + number2 + " = " + answer + " is " +
18       (number1 + number2 == answer));
19   }
20 }
```

**LISTING 5.1** RepeatAdditionQuiz.java

```
11     System.out.print(
12       "What is " + number1 + " + " + number2 + "? ");
13     int answer = input.nextInt();
14
15     while (number1 + number2 != answer) {
16       System.out.print("Wrong answer. Try again. What is "
17         + number1 + " + " + number2 + "? ");
18       answer = input.nextInt();
19     }
20
21     System.out.println("You got it!");
22   }
23 }
```

# The while loop

- Note that in the previous example, we don't know how many times the loop body will get executed.
- Lines 11-13 have to be repeated in the while loop (lines 16-18).
- If the first input for line 13 is already the correct answer, the body of the while loop is executed 0 time.

```
11      System.out.print(
12        "What is " + number1 + " + " + number2 + "? ");
13      int answer = input.nextInt();
14
15      while (number1 + number2 != answer) {
16        System.out.print("Wrong answer. Try again. What is "
17          + number1 + " + " + number2 + "? ");
18        answer = input.nextInt();
19      }
20
21      System.out.println("You got it!");
22    }
23  }
```



# A case study on guessing numbers

- You will 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 make the next guess intelligently. Here is a sample run:

```
Guess a magic number between 0 and 100
Enter your guess: 50 Enter
Your guess is too high
Enter your guess: 25 Enter
Your guess is too low
Enter your guess: 42 Enter
Your guess is too high
Enter your guess: 39 Enter
Yes, the number is 39
```

# A case study on guessing numbers (cont'd)

- It is important to *think before coding.*
- It is a good practice to *code incrementally* one step at a time.
  (1) Generate a random number between **0** and **100**, inclusive;
  (2) Then prompt the user to enter a guess;
  (3) Next is to compare the guess with the random number.
- For programs involving loops, if you don't know how to write a loop right away, you may first write the code for executing it one time, and then figure out how to repeatedly execute the code in a loop.
- For this program, you may create an initial draft, as in Listing 5.2. It prompts the user to enter a guess **only once**.

**LISTING 5.2** GuessNumberOneTime.java

```
1  import java.util.Scanner;
2
3  public class GuessNumberOneTime {
4    public static void main(String[] args) {
5      // Generate a random number to be guessed
6      int number = (int)(Math.random() * 101);
7
8      Scanner input = new Scanner(System.in);
9      System.out.println("Guess a magic number between 0 and 100");
10
11     // Prompt the user to guess the number
12     System.out.print("\nEnter your guess: ");
13     int guess = input.nextInt();
14
15     if (guess == number)
16       System.out.println("Yes, the number is " + number);
17     else if (guess > number)
18       System.out.println("Your guess is too high");
19     else
20       System.out.println("Your guess is too low");
21   }
22 }
```



**LISTING 5.2** GuessNumberOneTime.java

```
1  import java.util.Scanner;
2
3  public class GuessNumberOneTime {
4    public static void main(String[] args) {
5      // Generate a random number to be guessed
6      int number = (int)(Math.random() * 101);
7
8      Scanner input = new Scanner(System.in);
9      System.out.println("Guess a magic number between 0 and 100");
10
11     // Prompt the user to guess the number
12     System.out.print("\nEnter your guess: ");
13     int guess = input.nextInt();
14
15     if (guess == number)
16       System.out.println("Yes, the number is " + number);
17     else if (guess > number)
18       System.out.println("Your guess is too high");
19     else
20       System.out.println("Your guess is too low");
21   }
22 }
```

The loop repeatedly prompts the user to enter a guess (lines 22 and 23, which are the same as lines 12 and 13), until **guess** matches **number**, then the loop should end.

```
while (guess != number) {
  // loop body
}
```

```
1  import java.util.Scanner;
2
3  public class GuessNumber {
4    public static void main(String[] args) {
5      // Generate a random number to be guessed
6      int number = (int)(Math.random() * 101);
7
8      Scanner input = new Scanner(System.in);
9      System.out.println("Guess a number between 0 and 100");
10
11     // Prompt the user to guess the number
12     System.out.print("\nEnter your guess: ");
13     int guess = input.nextInt();
14
15     while (guess != number) {
16       if (guess > number)
17         System.out.println("Your guess is too high");
18       else
19         System.out.println("Your guess is too low");
20
21       // Prompt the user to guess the number again
22       System.out.print("\nEnter your guess: ");
23       guess = input.nextInt();
24     } // End of loop
25
26     // loop is terminated
27     System.out.println("Yes, the number is " + number);
28   }
29 }
```

What is the minimum number of times this loop here gets executed?

# Loop Design Strategies

- Writing a correct loop is not an easy task for beginners. Consider three steps when writing a loop.

  Step 1: Identify the statements that need to be repeated.

  Step 2: Wrap these statements in a loop like this:

```
while (true) {
    Statements;
}
```

  Step 3: Code the **loop-continuation-condition** and add appropriate statements for controlling the loop.

```
while (loop-continuation-condition) {
    Statements;
    Additional statements for controlling the loop;
}
```

- Rewrite Listing 3.3, SubtractionQuiz.java so that it generates five questions. Follow the loop design strategy.
  - First identify the statements that need to be repeated. These are the statements for obtaining two random numbers, prompting the user with a subtraction question, and grading the question.
  - Second, wrap the statements in a loop.
  - Third, add a loop control variable and the **loop-continuation-condition** to execute the loop five times.
  - **LISTING 5.4** SubtractionQuizLoop.java is a solution to this requirement.



# Controlling a Loop with a Sentinel Value

- Another common technique for controlling a loop is to designate a special value when reading and processing a set of values. This special input value, known as a *sentinel value*, signifies the end of the input.
- A loop that uses a sentinel value to control its execution is called a *sentinel-controlled loop*.
- Listing 5.5 writes a program that reads and calculates the sum of an unspecified number of integers. The input **0** signifies the end of the input.
- Note that if the first input read is **0**, the loop body never executes, and the resulting sum is **0**.

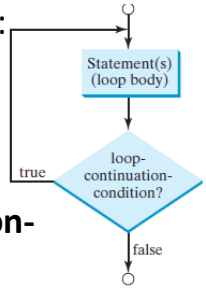
**LISTING 5.5** SentinelValue.java

```
1  import java.util.Scanner;
2
3  public class SentinelValue {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Read an initial data
10     System.out.print(
11       "Enter an integer (the input ends if it is 0): ");
12     int data = input.nextInt();
13
14     // Keep reading data until the input is 0
15     int sum = 0;
16     while (data != 0) {
17       sum += data;
18
19       // Read the next data
20       System.out.print(
21         "Enter an integer (the input ends if it is 0): ");
22       data = input.nextInt();
23     }
24
25     System.out.println("The sum is " + sum);
26   }
27 }
```

# The do-while Loop

- *A* **do-while** *loop is the same as a* **while** *loop except that it executes the loop body first and then checks the loop continuation condition.*
- The **do**-**while** loop is a variation of the **while** loop. Its syntax is:

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

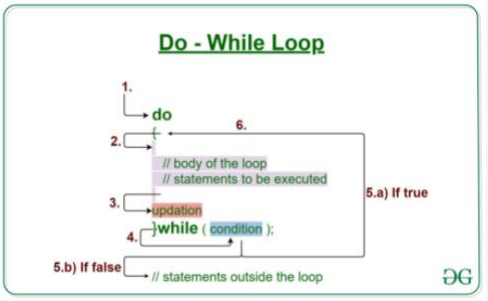



- The loop body is executed first, and then the **loop-continuation-condition** is evaluated.
- Use a **do-while** loop if you have statements inside the loop that must be executed *at least once*.




- **while** loops
- **do-while** loops
- **for** loops


# The do-while Loop




Reference: https://www.geeksforgeeks.org/java-do-while-loop-with-examples/?ref=lbp

- Write a do while loop that keeps asking user for input until -999 to stop.

- **while** loops
- **do-while** loops
- **for** loops

# The do-while Loop: an example

- Rewrite the **while** loop in Listing 5.5 using a **do-while** loop, as shown in Listing 5.6. Which one is a more convenient choice and why?

**LISTING 5.5** SentinelValue.java

```
1  import java.util.Scanner;
2
3  public class SentinelValue {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Read an initial data
10     System.out.print(
11       "Enter an integer (the input ends if it is 0): ");
12     int data = input.nextInt();
13
14     // Keep reading data until the input is 0
15     int sum = 0;
16     while (data != 0) {
17       sum += data;
18
19       // Read the next data
20       System.out.print(
21         "Enter an integer (the input ends if it is 0): ");
22       data = input.nextInt();
23     }
24
25     System.out.println("The sum is " + sum);
26   }
27 }
```

**LISTING 5.6** TestDoWhile.java

```
1  import java.util.Scanner;
2
3  public class TestDoWhile {
4    /** Main method */
5    public static void main(String[] args) {
6      int data;
7      int sum = 0;
8
9      // Create a Scanner
10     Scanner input = new Scanner(System.in);
11
12     // Keep reading data until the input is 0
13     do {
14       // Read the next data
15       System.out.print(
16         "Enter an integer (the input ends if it is 0): ");
17       data = input.nextInt();
18
19       sum += data;
20     } while (data != 0);
21
22     System.out.println("The sum is " + sum);
23   }
24 }
```



- **while** loops
- **do-while** loops
- **for** loops

# The for Loop

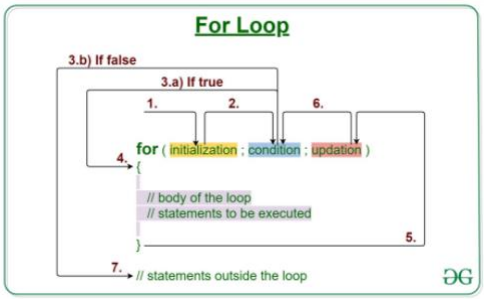
- *A* **for** *loop has a concise syntax for writing loops.* The syntax is:

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

- 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.

- **while** loops
- **do-while** loops
- **for** loops

# The for Loop




Reference: https://www.geeksforgeeks.org/java-for-loop-with-examples/

- Write a for loop that prints 0 to 100.
- Write a for loop that goes through a string character by character.




- **while** loops
- **do-while** loops
- **for** loops


# The for Loop

Step 1: initialize counter

Step 2: evaluate test

```
for (int i = 0; i <= 100; i++) {
   // Loop body
   ...
}
```

Step 3: execute loop body if test result is ture. False, quit loop.

Step 4: update counter

- **while** loops
- **do-while** loops
- **for** loops

# The for Loop

- Often you write a loop in the following common form:

```
for (int i = initialValue; i < endValue; i++) {
  // Loop body
  ...
}
```

equivalent to

```
int i = initialValue; // Initialize loop control variable
while (i < endValue)
  // Loop body
  ...
  i++; // Adjust loop control variable
}
```

- The control variable (that is variable i as in the code above) must be declared inside the control structure of the loop or before the loop.
- If the loop control variable is used only in the loop, and not elsewhere, it is a good programming practice to declare it in the **initial-action** of the **for** loop. For example: **for** (**int** i = **0**; i < **100**; i++) {…}
- If the variable is declared inside the loop control structure, it cannot be referenced outside the loop.



- **while** loops
- **do-while** loops
- **for** loops

# The for Loop: flowchart

- The flowchart of the **for** loop is shown in Figure 5.3a
- Write the code for the flowchart in Figure 5.3b and write a sentence to state what the for loop does.

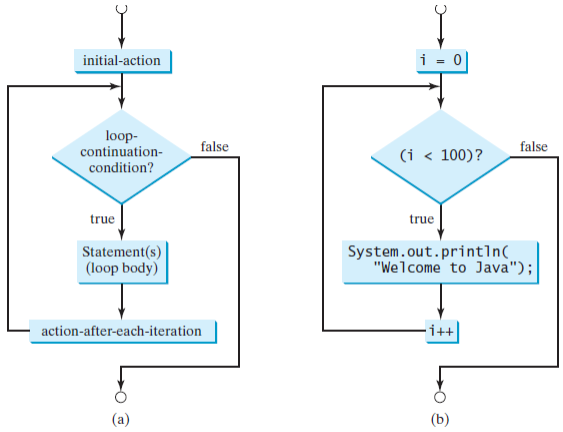



**FIGURE 5.3** A `for` loop performs an initial action once, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the `loop-continuation-condition` evaluates to `true`.

- **while** loops
- **do-while** loops
- **for** loops

# The for Loop: some additional notes

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

```
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. For example:

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

  - This example is correct, but it is a bad example, because it makes the code difficult to read. Normally, you declare and initialize a control variable as an initial action and increment or decrement the control variable as an action after each iteration.



# Which loop to use?

- *You can use a* **for** *loop, a* **while** *loop, or a* **do-while** *loop, whichever is convenient.*
- The **while** loop and **for** loop are called *pretest loops* because the continuation condition is checked before the loop body is executed.
- The **do-while** loop is called a *posttest loop* because the condition is checked after the loop body is executed.
- The three forms of loop statements—**while**, **do-while**, 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 **for** loop in (b).

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

(a)

Equivalent

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

(b)

# Which loop to use? (cont'd)

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

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

(a)

Equivalent

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

(b)

- In general, a **for** loop may be used if the number of repetitions is known in advance, as, for example, when you need to display a message a hundred times.
- A **while** loop may be used if the number of repetitions is not fixed, 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 the continuation condition is tested.



# Nested loops

- *A loop can be nested inside another loop.*
- *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 started anew.

Outer loop

```
for (int i = 1; i < 5; i++) {
    for (int j = 1; j < 5; j++) {
        System.out.print(i * j);
        System.out.print(' ');
    }
}
```

Inner loop

| Outer for loop for (int i = 1; i < 5; i++) | Inner for loop for (int j = 1; j < 5; j++ ) | System.out.print(i * j); |
|---|---|---|
| When i = 1 | j = 1 | 1 |
| | j = 2 | 2 |
| | j = 3 | 3 |
| | j = 4 | 4 |
| When i = 2 | j = 1 | 2 |
| | j = 2 | 4 |
| | j = 3 | 6 |
| | j = 4 | 8 |
| When i = 3 | j = 1 | 3 |
| | j = 2 | 6 |
| | j = 3 | 9 |
| | j = 4 | 12 |
| When i = 4 | j = 1 | 4 |
| | j = 2 | 8 |
| | j = 3 | 12 |
| | j = 4 | 16 |

# Nested Loops example

- Listing 5.7 presents a program that uses nested **for** loops to display a multiplication table.
- The program displays a title (line 5) on the first line in the output.
- The first **for** loop (lines 9–10) displays the numbers **1** through **9** on the second line.
- A dashed (-) line is displayed on the third line (line 12).

**LISTING 5.7** MultiplicationTable.java

```
1  public class MultiplicationTable {
2    /** Main method */
3    public static void main(String[] args) {
4      // Display the table heading
5      System.out.println("           Multiplication Table");
6
7      // Display the number title
8      System.out.print("    ");
9      for (int j = 1; j <= 9; j++)
10       System.out.print("   " + j);
11
12     System.out.println("\n--------------------------------------");
13
14     // Display table body
15     for (int i = 1; i <= 9; i++) {
16       System.out.print(i + " | ");
17       for (int j = 1; j <= 9; j++) {
18         // Display the product and align properly
19         System.out.printf("%4d", i * j);
20       }
21       System.out.println();
22     }
23   }
24 }
```

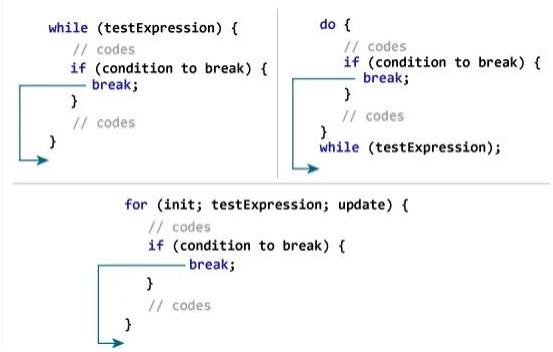
```
           Multiplication Table
    1   2   3   4   5   6   7   8   9
--------------------------------------
1 |   1   2   3   4   5   6   7   8   9
2 |   2   4   6   8  10  12  14  16  18
3 |   3   6   9  12  15  18  21  24  27
4 |   4   8  12  16  20  24  28  32  36
5 |   5  10  15  20  25  30  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
```

Try to trace the program in debugger of Eclipse.



# Keyword *break*

- *The* **break** *and* **continue** *keywords provide additional controls in a loop.*
- You have used the keyword **break** in a **switch** statement. You can also use **break** in a loop to immediately terminate the loop.




Reference: https://www.programiz.com/java-programming/break-statement

# Keyword *break:* example

- Listing 5.12 presents a program to demonstrate the effect of using **break** in a loop.
- The program in Listing 5.12 adds integers from **1** to **20** in this order to **sum** until **sum** is greater than or equal to **100**. Without the **if** statement (line 9), the program calculates the sum of the numbers from **1** to **20**.
- But with the **if** statement, the loop terminates when **sum** becomes greater than or equal to **100**.

**LISTING 5.12** TestBreak.java

```
1  public class TestBreak {
2    public static void main(String[] args) {
3      int sum = 0;
4      int number = 0;
5
6      while (number < 20) {
7        number++;
8        sum += number;
9        if (sum >= 100)
10         break;     // quit loop
11     }
12
13     System.out.println("The number is " + number);
14     System.out.println("The sum is " + sum);
15   }
16 }
```

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



# Keyword *continue*

- You can also use the **continue** keyword in a loop.
- When it is encountered, it ends the current iteration and program control goes to the end of the loop body.
- In other words, **continue** breaks out of an iteration while the **break** keyword breaks out of a loop.

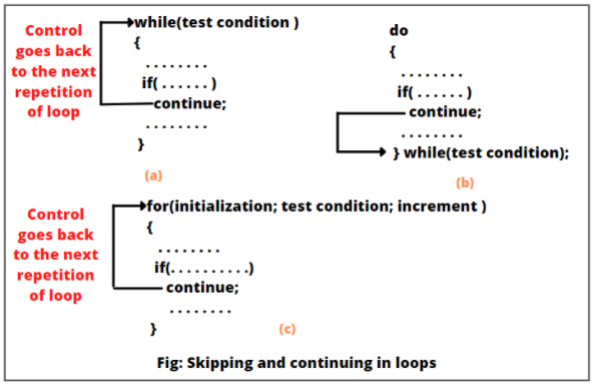



**Fig: Skipping and continuing in loops**

Reference: https://www.scientecheasy.com/2021/05/continue-statement-in-java.html/

# Keyword *continue:* example

- Listing 5.13 presents a program to demonstrate the effect of using **continue** in a loop.
- The program in Listing 5.13 adds integers from **1** to **20** except **10** and **11** to **sum**.
- When **number** becomes **10** or **11,** 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**.

**LISTING 5.13** TestContinue.java

```
1  public class TestContinue {
2    public static void main(String[] args) {
3      int sum = 0;
4      int number = 0;
5
6      while (number < 20) {
7        number++;
8        if (number ==10 || number == 11)
9          continue;
10       sum += number;
11     }
12
13     System.out.println("The sum is " + sum);
14   }
15 }
```

```
The sum is 189
```



# Keyword *break:* with or without

- You can always write a program without using **break** or **continue** in a loop.
- Suppose you need to write a program to find the smallest factor other than **1** for an integer **n** (assume **n >= 2**).

```
int factor = 2;
while (factor <= n) {
  if (n % factor == 0)
    break;
  factor++;
}
System.out.println("The smallest factor other than 1 for "
  + n + " is " + factor);
```

```
boolean found = false;
int factor = 2;
while (factor <= n && !found) {
  if (n % factor == 0)
    found = true;
  else
    factor++;
}
System.out.println("The smallest factor other than 1 for "
  + n + " is " + factor);
```

Programming is a creative endeavor. There are many different ways to write code. In fact, you can find a smallest factor using a rather simple code as follows:

```
int factor = 2;
while (factor <= n && n % factor != 0)
  factor++;
```

# Reading data from a file using **Scanner**

- The **java.util.Scanner** class was used to read strings and primitive values from the console in Section 2.3, Reading Input from the Console.
- A **Scanner** breaks its input into tokens delimited by whitespace characters. To read from the keyboard, you create a **Scanner** for **System.in**, as follows:
  Scanner input = **new** Scanner(System.in);
- To read from a file, create a **Scanner** for a file, as follows:
  Scanner input = **new** Scanner(**new** File(filename));

| java.util.Scanner | |
|---|---|
| +Scanner(source: File) | Creates a Scanner that scans tokens from the specified file. |
| +Scanner(source: String) | Creates a Scanner that scans tokens from the specified string. |
| +close() | Closes this scanner. |
| +hasNext(): boolean | Returns true if this scanner has more data to be read. |
| +next(): String | Returns next token as a string from this scanner. |
| +nextLine(): String | Returns a line ending with the line separator from this scanner. |
| +nextByte(): byte | Returns next token as a byte from this scanner. |
| +nextShort(): short | Returns next token as a short from this scanner. |
| +nextInt(): int | Returns next token as an int from this scanner. |
| +nextLong(): long | Returns next token as a long from this scanner. |
| +nextFloat(): float | Returns next token as a float from this scanner. |
| +nextDouble(): double | Returns next token as a double from this scanner. |
| +useDelimiter(pattern: String): Scanner | Sets this scanner's delimiting pattern and returns this scanner. |

**FIGURE 12.9** The Scanner class contains the methods for scanning data.



# Reading data from a file using **Scanner: example**

```
1  import java.util.Scanner;
2
3  public class ReadFileData {
4      public static void main(String[] args) throws Exception{
5          // TODO Auto-generated method stub
6          java.io.File inputFile = new java.io.File("input.txt");
7          Scanner input = new Scanner(inputFile);
8          int sum = 0;
9          // Keep reading data until no more input in the file
10         while (input.hasNext()) {
11                 sum += input.nextInt();
12         }
13         System.out.println("The sum is " + sum);
14     }
15 }
```

Try running this program without the existence of "input.txt" to see what happens.

Suppose the contents of the file are as follows:
2 3 4 5 6 7 8 9 12 23 32
23 45 67 89 92 12 34 35 3 1 2 4 0
The program should get **sum** to be **518**.

- The Scanner continues to read until it finds an "end of file" condition.
- You only get 'false' for hasNext() when you have a EOF (end-of-file) character.
- Invoking the constructor **new Scanner(File)** may throw an I/O exception, so the **main** method declares **throws Exception** in line 4.
  One possibility is attempting to create a Scanner object for a nonexistent file.

# Testing for valid input

- A Scanner has methods to see what the next token will be.

| Method | Description |
|---|---|
| `hasNext()` | returns `true` if there are any more tokens of input to read *(always true for console input)* |
| `hasNextInt()` | returns `true` if there is a next token and it can be read as an `int` |
| `hasNextDouble()` | returns `true` if there is a next token and it can be read as a `double` |
| `hasNextLine()` | returns `true` if there are any more lines of input to read *(always true for console input)* |

- These methods do not consume input. They give information about what the input is waiting.



# Chapter Summary

- There are three types of repetition statements: the **while** loop, the **do-while** loop, and the **for** loop.
- The part of the loop that contains the statements to be repeated is called the *loop body.*
- A one-time execution of a loop body is referred to as an *iteration of the loop.*
- An *infinite loop* is a loop statement that executes infinitely.
- In designing loops, you need to consider both the *loop control structure* and the loop body.
- The **while** loop checks the **loop-continuation-condition** first. If the condition is **true**, the loop body is executed; if it is **false**, the loop terminates.
- The **do-while** loop is similar to the **while** loop, except that the **do-while** loop executes the loop body first and then checks the **loop-continuation-condition** to decide whether to continue or to terminate.

# Chapter Summary

- The **while** loop and the **do-while** loop often are used when the number of repetitions is not predetermined.
- A *sentinel value* is a special value that signifies the end of the loop.
- The **for** loop generally is used to execute a loop body a fixed number of times.
- The **while** loop and **for** loop are called *pretest loops* because the continuation condition is checked before the loop body is executed.
- The **do-while** loop is called a *posttest loop* because the condition is checked after the loop body is executed.
- Two keywords, **break** and **continue**, can be used in a loop.
- The **break** keyword immediately ends the innermost loop, which contains the break.
- The **continue** keyword only ends the current iteration.



# Exercises

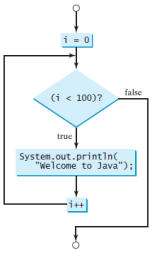
- Analyze the following code. Is count < 100 always true, always false, sometimes true and sometimes false at Point A, Point B, and Point C?

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

- How many times are the following loop bodies repeated?

```
int i = 1;
while (i < 10)
  if (i % 2 == 0)
    System.out.println(i);
```

# Exercises




- Write the corresponding java code for the following flowchart.

- Convert the following for loop statement to a while loop.

```
long sum = 0;
for (int i = 0; i <= 1000; i++)
sum = sum + i;
```

- Will the following programs terminate? If yes, what is the output.

```
int balance = 10;
while (true) {
  if (balance < 9)
    break;
  balance = balance - 9;
}

System.out.println("Balance is "
  + balance);
```

(a)

```
int balance = 10;
while (true) {
  if (balance < 9)
    continue;
  balance = balance - 9;
}

System.out.println("Balance is "
  + balance);
```

(b)