CMPE 101
Object Oriented Programming

İstanbul
Bilgi University

Dr. Emel Küpçü

Department of Computer Engineering

İstanbul Bilgi University

Week-4: Looping Statements, Loop Control and Logical Operators

do…while Iteration Statement

- The `do…while` **iteration statement** is similar to the `while` statement.
- In the `while`, the program tests the loop-continuation condition at the *beginning* of the loop, *before* executing the loop's body; if the condition is *false*, the body *never* executes.
- The do…`while` statement tests the loop-continuation condition *after* executing the loop's body; therefore, *the body always executes at least once*.
- When a do…`while` statement terminates, execution continues with the next statement in sequence.

```
1  // Fig. 5.7: DoWhileTest.java
2  // do...while iteration statement.
3
4  public class DoWhileTest {
5     public static void main(String[] args) {
6        int counter = 1;
7
8        do {
9           System.out.printf("%d  ", counter);
10          ++counter;
11       } while (counter <= 10);
12
13       System.out.println();
14    }
15 }
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.7 | do...while iteration statement.

5.3 for Iteration Statement

- The general format of the `for` statement is

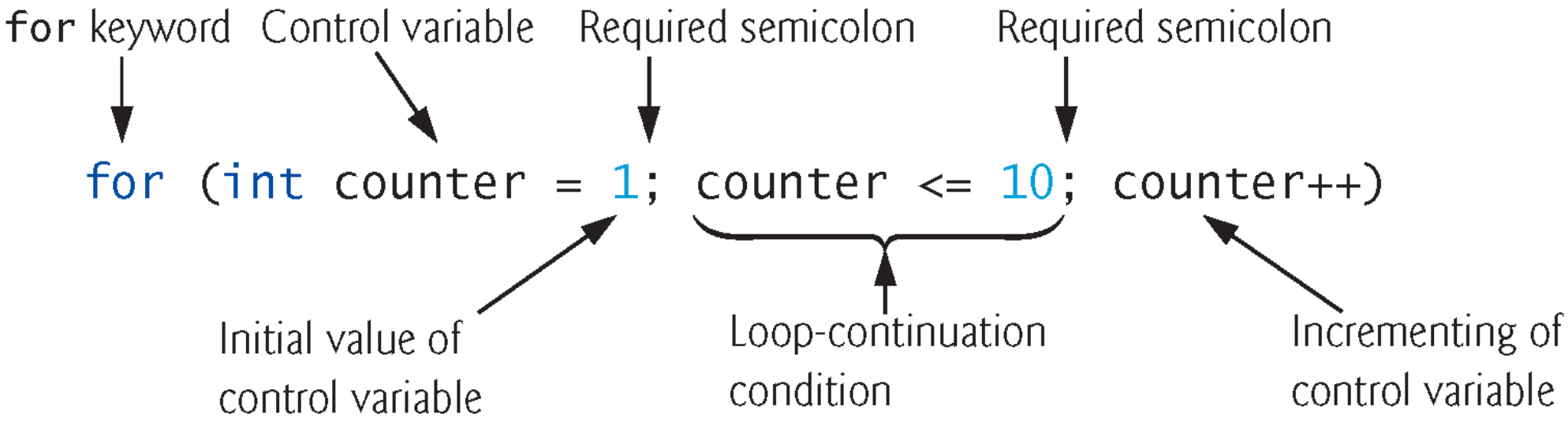
```
for (initialization; loopContinuationCondition; increment) {
    statement
}
```

  - The ***initialization*** expression names the loop's control variable and optionally provides its initial value.
  - ***loopContinuationCondition*** determines whether the loop should continue executing.
  - ***increment*** modifies the control variable's value, so that the loop-continuation condition eventually becomes false.
  - The **two semicolons** in the `for` header are **required**.

for Iteration Statement (Cont.)

Specifies the counter-controlled-iteration details in a single line of code.

- When the `for` statement begins executing, the control variable is *declared* and *initialized*.
- Next, the program checks the loop-continuation condition, which is between the two required semicolons.
- If the condition initially is true, the body statement executes.
- After executing the loop's body, the program increments the control variable in the increment expression, which appears to the right of the second semicolon.
- Then the loop-continuation test is performed again to determine whether the program should continue with the next iteration of the loop.



for Iteration Statement (Cont.)

```
for (initialization; loopContinuationCondition; increment)
    statement;
```

- The `for` statement often can be represented with an equivalent **`while`** statement as follows:

```
initialization;
while (loopContinuationCondition)
{
   statement;
   increment;
}
```

```
1  // Fig. 5.1: WhileCounter.java
2  // Counter-controlled iteration with the while iteration statement.
3
4  public class WhileCounter {
5     public static void main(String[] args) {
6        int counter = 1; // declare and initialize control variable
7
8        while (counter <= 10) { // loop-continuation condition
9           System.out.printf("%d  ", counter);
10          ++counter; // increment control variable
11       }
12
13       System.out.println();
14    }
15 }
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.1 | Counter-controlled iteration with the `while` iteration statement.

```
1  // Fig. 5.2: ForCounter.java
2  // Counter-controlled iteration with the for iteration statement.
3
4  public class ForCounter {
5     public static void main(String[] args) {
6        // for statement header includes initialization,
7        // loop-continuation condition and increment
8        for (int counter = 1; counter <= 10; counter++) {
9           System.out.printf("%d  ", counter);
10       }
11
12       System.out.println();
13    }
14 }
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 5.2 | Counter-controlled iteration with the `for` iteration statement.

for Iteration Statement (Cont.)

- All three expressions in a `for` header are optional.
  - If the *loopContinuationCondition* is omitted, the condition is always true, thus creating an infinite loop.
  - You might omit the *initialization* expression if the program initializes the control variable before the loop.
  - You might omit the *increment* if the program calculates it with statements in the loop's body or if no increment is needed.
- In a for loop, the increment step runs at the end of each iteration, just like a separate statement.
- ```
  counter = counter + 1
  counter += 1
  ++counter
  counter++
  ```
  are **equivalent** increment expressions in a `for` statement.
  ```

for Iteration Statement (Cont.)

- The initialization, loop-continuation condition and increment can contain arithmetic expressions.
- For example, assume that x = 2 and y = 10. If x and y are not modified in the body of the loop, the statement

```
for (int j = x; j <= 4 * x * y; j += y / x)
```

- is equivalent to the statement

```
for (int j = 2; j <= 80; j += 5)
```

- The increment of a `for` statement may be ***negative***, in which case it's a **decrement**, and the loop counts ***downward***.

Examples Using the for Statement

- a)Vary the control variable from 1 to 100 in increments of 1.

```
for (int i = 1; i <= 100; i++)
```

- b)Vary the control variable from 100 to 1 in decrements of 1.

```
for (int i = 100; i >= 1; i--)
```

- c)Vary the control variable from 7 to 77 in increments of 7.

- d)Vary the control variable from 20 to 2 in decrements of 2.

- e)Vary the control variable over the values 2, 5, 8, 11, 14, 17, 20.

- f)Vary the control variable over the values 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

Common Programming Error 5.6

Do not use equality operators (`!=` or `==`) in a loop-continuation condition if the loop's control variable increments or decrements by more than 1. For example, consider the `for` statement header:

```
for (int counter = 1; counter != 10; counter += 2)
```

The loop-continuation test `counter != 10` never becomes `false` (resulting in an infinite loop) because `counter` increments by 2 after each iteration.

Example Using the for Statement

```
1  // Fig. 5.5: Sum.java
2  // Summing integers with the for statement.
3
4  public class Sum {
5     public static void main(String[] args) {
6        int total = 0;
7
8        // total even integers from 2 through 20
9        for (int number = 2; number <= 20; number += 2) {
10          total += number;
11       }
12
13       System.out.printf("Sum is %d%n", total);
14    }
15 }
```

```
Sum is 110
```

Fig. 5.5 | Summing integers with the `for` statement.

5.6 switch Multiple-Selection Statement

- `switch` **multiple-selection statement** performs different actions based on the possible values of a **constant integral expression** of type `byte`, `short`, `int` or `char`.

- **General Format:**

```
switch (variable) {
   case value1:
      // code to execute if variable == value1
      break;
   case value2:
      // code to execute if variable == value2
      break;
   // additional cases
   default:
      // code to execute if no cases match
}
```

switch Multiple-Selection Statement (cont.)

```
public class DayOfWeek {
    public static void main(String[] args) {
        int day = 5;
        String dayName;

        switch (day) {
            case 1:
                dayName = "Sunday";
                break;
            case 2:
                dayName = "Monday";
                break;
            case 3:
                dayName = "Tuesday";
                break;
            case 4:
                dayName = "Wednesday";
                break;
            case 5:
                dayName = "Thursday";
                break;
            case 6:
                dayName = "Friday";
                break;
            case 7:
                dayName = "Saturday";
                break;
            default:
                dayName = "Invalid day";
                break;
        }
        System.out.println("Day of the week: " + dayName);
    }
}
```

Output:
Day of the week: Thursday

5.6 `switch` Multiple-Selection Statement (Cont.)

- The `switch` statement consists of a block that contains a sequence of `case` **labels** and an optional `default` **case.**
- The program evaluates the **controlling expression** in the parentheses following keyword `switch`.
- The program checks if the given value (which must be a byte, char, short, int, or String) matches any case label.
- If a match occurs, the program executes that `case`'s statements.
- The **break** statement exits the switch and continues with the next code.

5.6 `switch` Multiple-Selection Statement (Cont.)

- `switch` does *not* provide a mechanism for testing ranges of values—**every value must be listed in a separate `case` label**.
- Note that each `case` can have multiple statements.
- `switch` differs from other control statements in that it does not require braces around multiple statements in a `case`.
- Without break, the code keeps running from the matching case to the next until it hits a break or the switch ends. This is called "falling through".
- If no match occurs between the controlling expression's value and a `case` label, the **`default`** case executes.
- if none of the cases match and there is no default case, the program skips the switch block and moves to the next line of code after it.

Common Programming Error 5.7

Forgetting a `break` statement when one is needed in a `switch` is a logic error.

Error-Prevention Tip 5.9

In a `switch` statement, ensure that you test all possible values of the controlling expression.

Error-Prevention Tip 5.10

Provide a `default` case in `switch` statements. This focuses you on the need to process exceptional conditions.

switch Multiple-Selection Statement (Cont.)

- In a switch statement, each case must use a fixed number or character.
- An integer constant is simply an integer value.
- In addition, you can use **character constants**—specific characters in single quotes, such as `'A'`, `'7'` or `'$'`—which represent the integer values of characters.

```
char grade = 'A'; // 'A' is a character constant
switch (grade) {
   case 'A':
      System.out.println("Excellent!");
      break;
   case 'B':
      System.out.println("Good!");
      break;
   default:
      System.out.println("Keep improving!");
}
```

Output:
Excellent!

switch Multiple-Selection Statement (Cont.)

- `Strings` can be used as controlling expressions in switch statements, and `String` literals can be used in case labels.

```
public class switchClass {
    public static void main(String[] args) {
        String day = "Monday";  // String as controlling expression

        switch (day) {
            case "Monday":
                System.out.println("Start of the work week!");
                break;
            case "Friday":
                System.out.println("Weekend is near!");
                break;
            case "Sunday":
                System.out.println("Relax, it's the weekend.");
                break;
            default:
                System.out.println("Just another day.");
        }
    }
}
```

Output:
Start of the work week!

5.8 break and continue Statements

- The **break** statement, when executed in a `while`, `for`, `do…while` or `switch`, causes immediate exit from that statement.
- Execution continues with the first statement after the control statement.
- Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch`.

```java
// Fig. 5.13: BreakTest.java
// break statement exiting a for statement.
public class BreakTest {
   public static void main(String[] args) {
      int count; // control variable also used after loop terminates

      for (count = 1; count <= 10; count++) { // loop 10 times
         if (count == 5) {
            break; // terminates loop if count is 5
         }

         System.out.printf("%d ", count);
      }

      System.out.printf("%nBroke out of loop at count = %d%n", count);
   }
}
```

```
1 2 3 4
Broke out of loop at count = 5
```

Fig. 5.13 | break statement exiting a for statement.

5.8 break and continue Statements (Cont.)

- The **`continue`** statement, when executed in a `while`, `for` or `do…while`, skips the remaining statements in the loop body and proceeds with the *next iteration* of the loop.
- In while and do...while loops, the loop condition is checked right after a continue statement runs.
- In a `for` statement, the increment expression executes, then the program evaluates the loop-continuation test.

```
1  // Fig. 5.14: ContinueTest.java
2  // continue statement terminating an iteration of a for statement.
3  public class ContinueTest {
4     public static void main(String[] args) {
5        for (int count = 1; count <= 10; count++) { // loop 10 times
6           if (count == 5) {
7              continue; // skip remaining code in loop body if count is 5
8           }
9
10          System.out.printf("%d ", count);
11       }
12
13       System.out.printf("%nUsed continue to skip printing 5%n");
14    }
15 }
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Fig. 5.14 | `continue` statement terminating an iteration of a `for` statement.

5.9 Logical Operators

- Java's **logical operators** enable you to form more complex conditions by combining simple conditions.
- The logical operators are
  - && (conditional AND)
  - || (conditional OR)
  - & (boolean logical AND)
  - | (boolean logical inclusive OR)
  - ^ (boolean logical exclusive OR)
  - ! (logical NOT).
- [*Note:* The &, | and ^ operators are also bitwise operators when they are applied to integral operands.]

Logical Operators (Cont.)

- The && (**conditional AND**) operator ensures that two conditions are *both true* before choosing a certain path of execution.
- The table in Fig. 5.15 summarizes the && operator. The table shows all four possible combinations of `false` and `true` values for *expression1* and *expression2.*
- Such tables are called **truth tables**. Java evaluates to `false` or `true` all expressions that include relational operators, equality operators or logical operators.

expression1	expression2	expression1 && expression2
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Fig. 5.15 | && (conditional AND) operator truth table.

Logical Operators (Cont.)

- The || (**conditional OR**) operator ensures that *either or both* of two conditions are true before choosing a certain path of execution.
- Figure 5.16 is a truth table for operator conditional OR (||).
- Operator && has a higher precedence than operator ||.
- Both operators associate from left to right.

expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 5.16 | || (conditional OR) operator truth table.

Logical Operators (Cont.)

- The parts of an expression containing && or || operators are evaluated only until it's known whether the condition is true or false.
  - In expressions with && or ||, evaluation stops once the result is determined.

- This feature of conditional AND and conditional OR expressions is called **short-circuit evaluation**.

Common Programming Error 5.8

In expressions using &&, a condition—we'll call this the dependent condition—may require another condition to be `true` for the dependent condition's evaluation to be meaningful. In this case, the dependent condition should be placed after the && operator to prevent errors. Consider the expression `(i != 0) && (10 / i == 2)`. The dependent condition `(10 / i == 2)` must appear after the && to prevent the possibility of division by zero.

Logical Operators (Cont.)

- The **boolean logical AND** (**&**) and **boolean logical inclusive OR (|)** operators are identical to the && and || operators, except that the & and | operators *always evaluate both of their operands* (i.e., they do not perform short-circuit evaluation).
  - The **&** and **|** operators work like **&&** and **||**, but they always evaluate both operands.

- This is useful if the right operand has a required **side effect**—a modification of a variable's value.

Logical Operators (Cont.)

- **Example-1**: The && (conditional AND) and || (conditional OR) operators **stop evaluating** as soon as the result is known. This is called **short-circuit evaluation**.
  - x > 0 is true, so Java **does not** check x / 0 == 1, preventing an error.

```
int x = 5;
if (x > 0 || x / 0 == 1) {  // The second part is never checked
    System.out.println("Valid");
}
```

Output:
Valid

- **Example-2**:The & (logical AND) and | (logical inclusive OR) operators **always** evaluate **both sides**, even if the first part already determines the result.
  - Java **still** checks x / 0 == 1, causing a division by zero error.

```
int x = 5;
if (x > 0 | x / 0 == 1) {  // Both sides are checked
    System.out.println("Valid");
}
```

Output:
Exception in thread "main" java.lang.ArithmeticExceptio n: / by zero

Error-Prevention Tip 5.11

For clarity, avoid expressions with side effects (such as assignments) in conditions. They can make code harder to understand and can lead to subtle logic errors.

Logical Operators (Cont.)

- A simple condition containing the **boolean logical exclusive OR** (^) operator is `true` *if and only if* one of its operands is `true` and the other is `false`.
- If both are `true` or both are `false`, the entire condition is `false`.
- Figure 5.17 is a truth table for the boolean logical exclusive OR operator (^).
- This operator is guaranteed to evaluate both of its operands.

expression1	expression2	expression1 ^ expression2
false	false	false
false	true	true
true	false	true
true	true	false

Fig. 5.17 | ^ (boolean logical exclusive OR) operator truth table.

Logical Operators (Cont.)

- The ! (**logical NOT**, also called **logical negation** or **logical complement**) operator "reverses" the meaning of a condition.
- The logical negation operator is a *unary* operator that has only one condition as an operand.
- The ! (logical negation) operator reverses a condition, making true false and false true.
- You can often avoid ! by rewriting the condition with a different comparison operator.
- Figure 5.18 is a truth table for the logical negation operator.

expression	!expression
`false`	`true`
`true`	`false`

Fig. 5.18 | ! (logical NOT) operator truth table.

Figure 5.19 produces the truth tables discussed in this section.
The %b **format specifier** displays the word "true" or the word "false" based on a `boolean` expression's value.

```
1  // Fig. 5.19: LogicalOperators.java
2  // Logical operators.
3
4  public class LogicalOperators {
5     public static void main(String[] args) {
6        // create truth table for && (conditional AND) operator
7        System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
8           "Conditional AND (&&)", "false && false", (false && false),
9           "false && true", (false && true),
10          "true && false", (true && false),
11          "true && true", (true && true));
12
13       // create truth table for || (conditional OR) operator
14       System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
15          "Conditional OR (||)", "false || false", (false || false),
16          "false || true", (false || true),
17          "true || false", (true || false),
18          "true || true", (true || true));
```

Fig. 5.19 | Logical operators. (Part 1 of 5.)

```
Conditional AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Conditional OR (||)
false || false: false
false || true: true
true || false: true
true || true: true
```

```
19
20          // create truth table for & (boolean logical AND) operator
21          System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
22             "Boolean logical AND (&)", "false & false", (false & false),
23             "false & true", (false & true),
24             "true & false", (true & false),
25             "true & true", (true & true));
26
27          // create truth table for | (boolean logical inclusive OR) operator
28          System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
29             "Boolean logical inclusive OR (|)",
30             "false | false", (false | false),
31             "false | true", (false | true),
32             "true | false", (true | false),
33             "true | true", (true | true));
```

Fig. 5.19 | Logical operators. (Part 2 of 5.)

```
Boolean logical AND (&)
false & false: false
false & true: false
true & false: false
true & true: true

Boolean logical inclusive OR (|)
false | false: false
false | true: true
true | false: true
true | true: true
```

```
34
35          // create truth table for ^ (boolean logical exclusive OR) operator
36          System.out.printf("%s%n%s: %b%n%s: %b%n%s: %b%n%s: %b%n%n",
37             "Boolean logical exclusive OR (^)",
38             "false ^ false", (false ^ false),
39             "false ^ true", (false ^ true),
40             "true ^ false", (true ^ false),
41             "true ^ true", (true ^ true));
42
43          // create truth table for ! (logical negation) operator
44          System.out.printf("%s%n%s: %b%n%s: %b%n", "Logical NOT (!)",
45             "!false", (!false), "!true", (!true));
46       }
47    }
```

Fig. 5.19 | Logical operators. (Part 3 of 5.)

```
Boolean logical exclusive OR (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

Logical NOT (!)
!false: true
!true: false
```

Fig. 5.19 | Logical operators. (Part 5 of 5.)

Group Discussion Question

Write a Java program that does the following:

- **Uses a for loop** to iterate numbers from 1 to 10.
- **Skips numbers that are divisible by 3 or 5** using the **continue** statement.
- **Stops the loop completely if the number is greater than 8** using the **break** statement.
- **Uses an else statement**, inside which:
  - A **switch** statement checks if the number is **even**.
  - If even, it prints "Even number: X", otherwise, it prints the number as is.
- What will be the output of the program?

Questions?