

Control Statement II

CS102A Lecture 4

James YU

yujq3@sustech.edu.cn

Department of Computer Science and Engineering
Southern University of Science and Technology

Sept. 28, 2020



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Objectives

- To use `for` and `while` statements.
- To use `switch` statement.
- To use `continue` and `break` statements.
- To use logical operators.

Counter-controlled repetition with **while**



```
1 public class WhileCounter {  
2     public static void main(String[] args) {  
3         int counter = 1; // Control variable (loop counter)  
4         while ( counter <= 10 ) { // Loop continuation condition  
5             System.out.printf("%d", counter);  
6             ++counter; // Counter increment (or decrement) in each iteration  
7         }  
8         System.out.println();  
9     }  
10 }
```

The **for** repetition statement

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

```
1 public class ForCounter {  
2     public static void main(String[] args) {  
3         for (int counter = 1; counter <= 10; counter++) {  
4             System.out.printf("%d", counter);  
5         }  
6         System.out.println();  
7     }  
8 }
```

Common logic error: Off-by-one



```
1 for(int counter = 0; counter < 10; counter++) {  
2     // loop how many times?  
3 }  
4 for(int counter = 0; counter <= 10; counter++) {  
5     // loop how many times?  
6 }  
7 for(int counter = 1; counter <= 10; counter++) {  
8     // loop how many times?  
9 }
```

The **for** and **while** loops

- In most cases, a **for** statement can be easily represented with an equivalent **while** statement.
- Typically, **for** statements are used for counter-controlled repetition and **while** statements for sentinel-controlled repetition.

Control variable scope in **for**



- If the initialization expression in the **for** header declares the control variable, the control variable can be used only in that **for** statement.

```
1 int i; // Declaration
```

- stating the type and name of a variable

```
1 i = 3; // Assignment
```

- storing a value in a variable

```
1 for(int i = 1; i <= 10; i++){  
2     // i can only be used  
3     // in the loop body  
4 }
```

```
1 int i;  
2 for(i = 1; i <= 10; i++){  
3     // i can be used here  
4 }  
5 // i can also be used  
6 // after the loop until  
7 // the end of the enclosing block
```

More on **for** Repetition Statement

- If the *loop-continuation condition* 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 no increment is needed.
- The *increment expression* in a **for** acts as if it were a standalone statement at the end of the *for*'s body, so

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

are equivalent increment expressions in a **for** statement.

More on **for** Repetition Statement

- The *initialization* and *increment/decrement expressions* can contain multiple expressions separated by commas.

```
1 for ( int number = 2; number <= 20; total += number, number += 2 )  
2   ; // empty statement
```

is equivalent to

```
1 for ( int number = 2; number <= 20; number += 2 ) {  
2   total += number;  
3 }
```



The **do...while** repetition statement

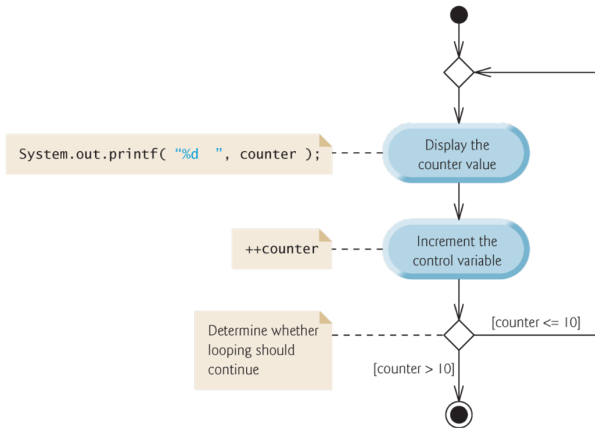
- **do...while** is like **while**.
- In **while**, the program tests the *loop-continuation condition* at the beginning of the loop, **before** executing the loop body; if the condition is **false**, the body never executes.
- **do...while** tests the *loop-continuation condition* **after** executing the loop body. The body always executes at least once.



Execution flow of `do...while`

```
1 int counter = 1;  
2 do {  
3     System.out.println(counter);  
4     ++counter;  
5 } while( counter <= 10 );
```

- Don't forget semicolon.



The **switch** multiple-selection statement



- The **switch** statement performs different actions based on the values of a *constant integral expression* of type **byte**, **short**, **int** or **char** etc.
- It consists of a block that contains a sequence of **case** labels and an optional **default** case.

```
1 switch (studentGrade) {  
2     case 'A':  
3         System.out.println("90 - 100");  
4         break;  
5     case 'B':  
6         System.out.println("80 - 89");  
7         break;  
8     case 'C':  
9         System.out.println("70 - 79");  
10        break;  
11     case 'D':  
12        System.out.println("60 - 69");  
13        break;  
14     default:  
15        System.out.println("score < 60"  
16        );  
16 }
```

The **switch** multiple-selection statement



- The program compares the *controlling expression*'s value with each **case** label.
- If a match occurs, the program executes that **case**'s statements.
- If no match occurs, the **default** case executes.
- If no match occurs and there is no **default** case, program simply **continues with the first statement after switch**.

```
1 switch (studentGrade) {  
2     case 'A':  
3         System.out.println("90 - 100");  
4         break;  
5     case 'B':  
6         System.out.println("80 - 89");  
7         break;  
8     case 'C':  
9         System.out.println("70 - 79");  
10        break;  
11     case 'D':  
12        System.out.println("60 - 69");  
13        break;  
14     default:  
15        System.out.println("score < 60"  
16        );  
16 }
```

The **switch** multiple-selection statement



- **switch** does not provide a mechanism for testing ranges of values — every value must be listed in a separate **case** label.
- Each **case** can have multiple statements (braces are optional).

```
1 switch (studentGrade) {  
2     case 90 <= studentGrade: // WRONG  
3         System.out.println("90 - 100");  
4         break;  
5     case 'B':  
6         System.out.println("80 - 89");  
7         break;  
8     case 'C':  
9         System.out.println("70 - 79");  
10        break;  
11     case 'D':  
12        System.out.println("60 - 69");  
13        break;  
14     default:  
15        System.out.println("score < 60"  
16        );  
17 }
```

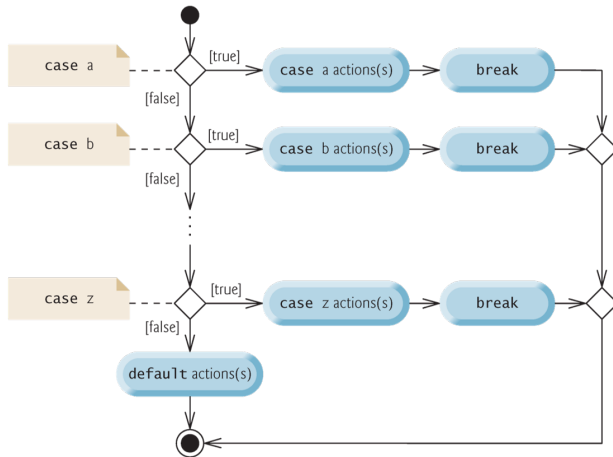
The **switch** multiple-selection statement



- *Falling through*: Without **break**, the statements for a matching **case** and subsequent **cases** execute until a **break** or the end of the switch is encountered.
 - If `studentGrade == 'A'`, then output is 90 -100 80 -89 70 -79

```
1 switch (studentGrade) {  
2     case 'A':  
3         System.out.println("90 - 100");  
4     case 'B':  
5         System.out.println("80 - 89");  
6     case 'C':  
7         System.out.println("70 - 79");  
8         break;  
9     case 'D':  
10        System.out.println("60 - 69");  
11        break;  
12    default:  
13        System.out.println("score < 60"  
14        );  
14 }
```

Execution flow of **switch**





The **break** statement

- 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**.

The **break** statement



```
1 // break statement exiting a for statement
2 public class BreakTest
3 {
4     public static void main(String[] args) {
5         int count; // control variable also used after loop terminates
6         for (count = 1; count <= 10; count++) { // loop 10 times
7             if (count == 5) // if count is 5
8                 break; // terminate loop
9             System.out.printf("%d ", count);
10        }
11        System.out.printf("\nBroke out of loop at count = %d\n", count);
12    }
13 }
```

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



The `continue` statement

- 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` statements, the program evaluates the loop-continuation test immediately after the `continue` statement executes.
- In a `for` statement, the *increment expression* executes, then the program evaluates the loop-continuation test.

The **continue** statement



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

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

Logical operators

- Logical operators help form complex conditions by combining simple ones:
 - `&&` (conditional AND)
 - `||` (conditional OR)
 - `&` (boolean logical AND)
 - `|` (boolean logical inclusive OR)
 - `^` (boolean logical exclusive OR)
 - `!` (logical NOT)
- `&`, `|` and `^` are also *bitwise operators* when applied to integral operands.

The **&&** (conditional **AND**) operator

- **&&** ensures that two conditions are both **true** before choosing a certain path of execution.
- 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

The `||` (conditional OR) operator

- `||` ensures that either or both of two conditions are `true` before choosing a certain path of execution.
- **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

Short-circuit evaluation of `&&` and `||`



- The expression containing `&&` or `||` operators are evaluated only until it's known whether the condition is `true` or `false`.

```
1 ( gender == FEMALE ) && ( age >= 65 )
```

Evaluation stops if the first part is `false`, the whole expression's value is `false`.

```
1 ( gender == FEMALE ) || ( age >= 65 )
```

Evaluation stops if the first part is `true`, the whole expression's value is `true`.

The `&` and `|` operators

- 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 (they do not perform short-circuit evaluation).
- This is useful if the right operand of the `&` or `|` has a required side effect — a modification of a variable's value.

```
1 int b = 0, c = 0;  
2 if(true || b == (c = 6)) System.out.println(c);
```

```
1 int b = 0, c = 0;  
2 if(true | b == (c = 6)) System.out.println(c);
```

The ^ operator

- 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**.
- This operator evaluates both of its operands.

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

The **!** (*logical NOT*) Operator

- **!** (a.k.a., *logical negation* or *logical complement*) unary operator “reverses” the value of a condition.

expression1	!expression1
false	true
true	false

The operators introduced so far



Operators						Associativity	Type
++	--					RTL	Unary postfix
++	--	+	-	!	(type)	RTL	Unary prefix
*	/	%				LTR	Multiplicative
+	-					LTR	Additive
<	<=	>	>=			LTR	Relational
==	!=					LTR	Equality
&						LTR	Boolean AND
^						LTR	Boolean XOR
						LTR	Boolean OR
&&						LTR	Conditional AND
						LTR	Conditional OR
?:						RTL	Conditional
=	+=	-=	*=	/=	%=	RTL	Assignment