Control Statement II

CS102A Lecture 4

James YU

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`

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

The for repetition statement

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

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

Common logic error: Off-by-one

```
for(int counter = 0; counter < 10; counter++) {
  // loop how many times?
}
for(int counter = 0; counter <= 10; counter++) {
  // loop how many times?
}
for(int counter = 1; counter <= 10; counter++) {
  // loop how many times?
}
```

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.

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

  - stating the type and name of a variable

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

  - storing a value in a variable

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

```
int i;
for(i = 1; i <= 10; i++){
  // i can be used here
}
// i can also be used
// after the loop until
// 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

```
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 expression*s can contain multiple expressions separated by commas.

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

is equivalent to

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

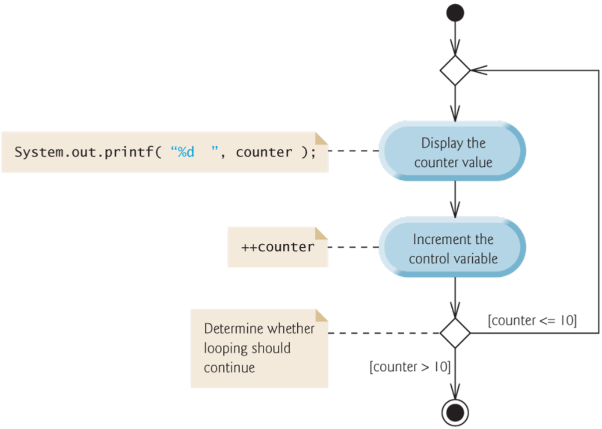
The `do...while` repetition statement

- `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`

```java
int counter = 1;
do {
  System.out.println(counter);
  ++counter;
} while( counter <= 10 );
```

- Don’t forget semicolon.



```java
System.out.printf( “%d  ”, counter );
++counter
```

Display the counter value
Increment the control variable
Determine whether looping should continue
[counter <= 10]
[counter > 10]

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.

```java
switch (studentGrade) {
  case 'A':
    System.out.println("90 - 100");
    break;
  case 'B':
    System.out.println("80 - 89");
    break;
  case 'C':
    System.out.println("70 - 79");
    break;
  case 'D':
    System.out.println("60 - 69");
    break;
  default:
    System.out.println("score < 60");
}
```

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**.

```java
switch (studentGrade) {
  case 'A':
    System.out.println("90 - 100");
    break;
  case 'B':
    System.out.println("80 - 89");
    break;
  case 'C':
    System.out.println("70 - 79");
    break;
  case 'D':
    System.out.println("60 - 69");
    break;
  default:
    System.out.println("score < 60");
}
```

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).

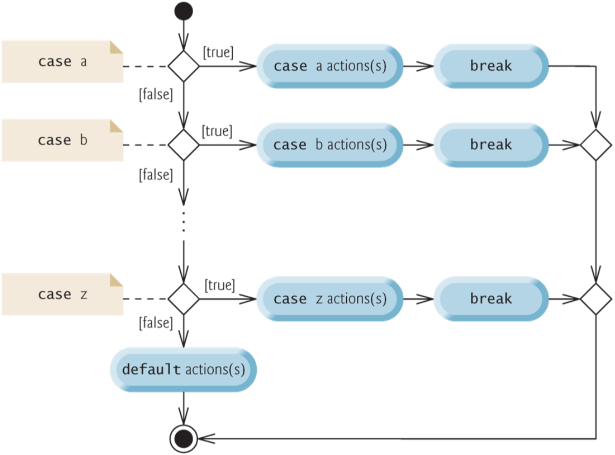
```
switch (studentGrade) {
  case 90 <= studentGrade: // WRONG
    System.out.println("90 - 100");
    break;
  case 'B':
    System.out.println("80 - 89");
    break;
  case 'C':
    System.out.println("70 - 79");
    break;
  case 'D':
    System.out.println("60 - 69");
    break;
  default:
    System.out.println("score < 60");
}
```

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

```
switch (studentGrade) {
  case 'A':
    System.out.println("90 - 100");
  case 'B':
    System.out.println("80 - 89");
  case 'C':
    System.out.println("70 - 79");
    break;
  case 'D':
    System.out.println("60 - 69");
    break;
  default:
    System.out.println("score < 60");
}
```

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

```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) // if count is 5
        break; // terminate loop
      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
```

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

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

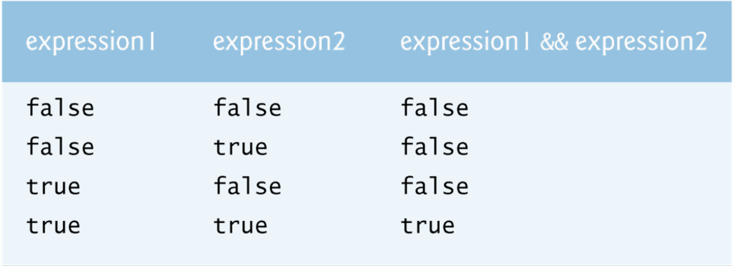
```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

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 operator*s when applied to integral operands.

The && (conditional AND) operator

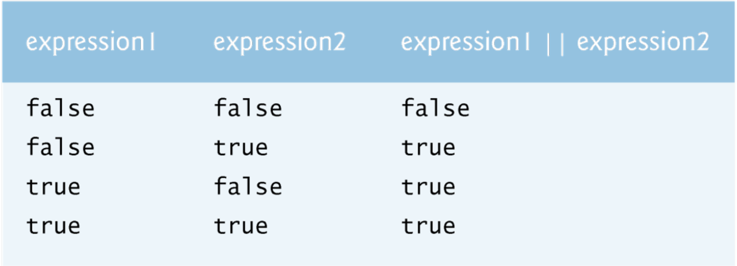
- && ensures that two conditions are both true before choosing a certain path of execution.
- Java evaluates to false or true all expressions that include relational operators, equality operators or logical operators.



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`.

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

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

```
( gender == FEMALE ) \code{|}\code{|} ( age >= 65 )
```

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

The & and | operators

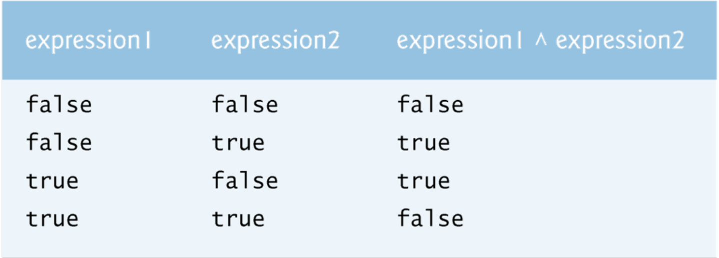
- The *boolean logical AND* (&) and *boolean logical inclusive OR* (|) operators are identical to the && and || operators, except that the & and | operators always evaluate both of their operands (they do not perform short-circuit evaluation).
- This is useful if the right operand of the & or | has a required side effect — a modification of a variable’s value.

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

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

The ^ operator

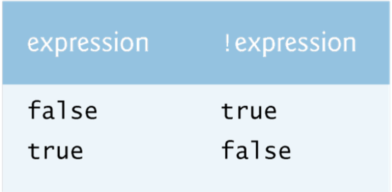
- A simple condition containing the *boolean logical exclusive OR* (`^`) operator is `true` if and only if one of its operands is `true` and the other is `false`.
- This operator evaluates both of its operands.



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

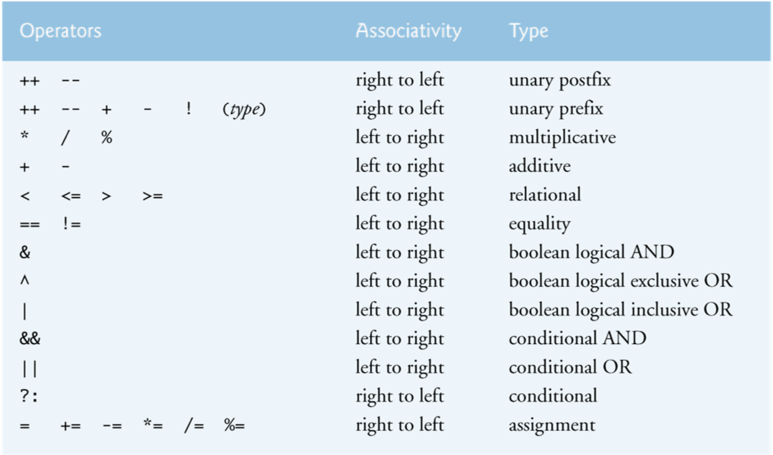
The ! (*logical NOT*) Operator

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



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

The operators introduced so far



Operators	Associativity	Type
++ --	right to left	unary postfix
++ -- + - ! (*type*)	right to left	unary prefix
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	boolean logical AND
^	left to right	boolean logical exclusive OR
|	left to right	boolean logical inclusive OR
&&	left to right	conditional AND
||	left to right	conditional OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment