Chapter 4: Control Statements (Part II)

Java™ How to Program, 11th Edition
Instructor: Zhuozhao Li

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`

```
public class WhileCounter {

    public static void main(String[] args) {

        int counter = 1;

        while ( counter <= 10 ) {

            System.out.printf("%d", counter);

            ++counter;

        }

        System.out.println();

    }

}
```

Control variable (loop counter)

Loop continuation condition

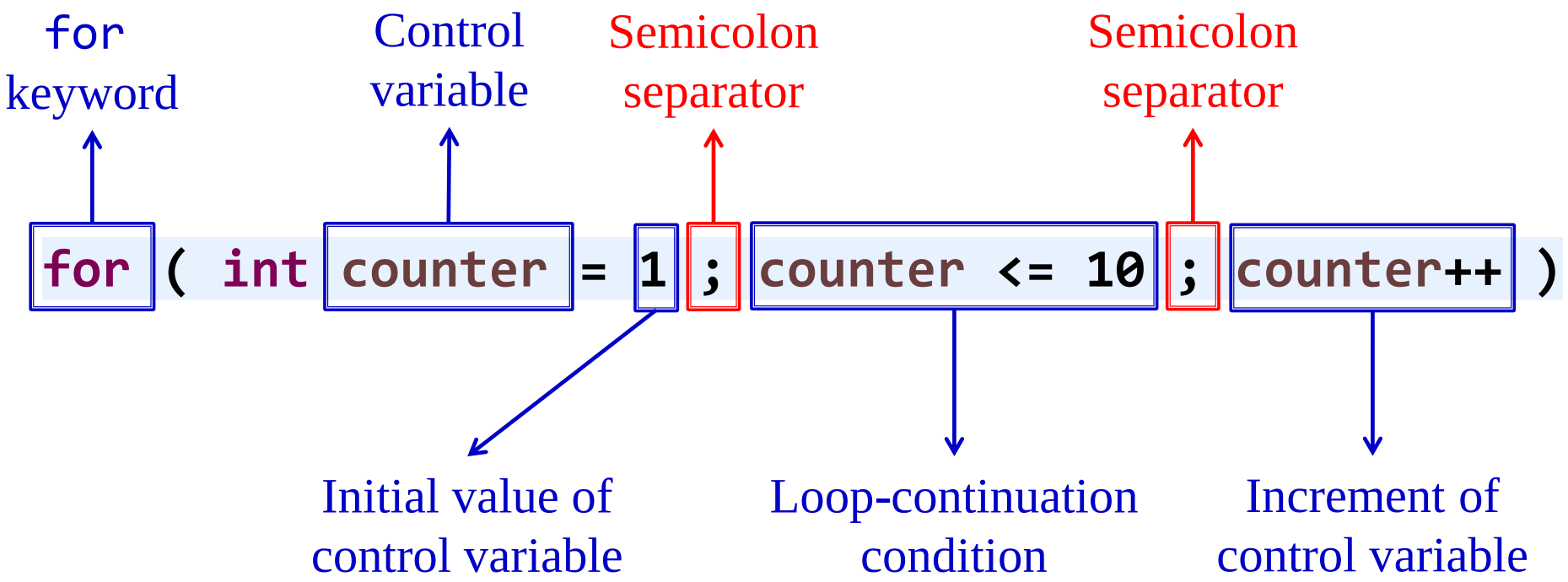
Counter increment (or decrement) in each iteration

The for Repetition Statement

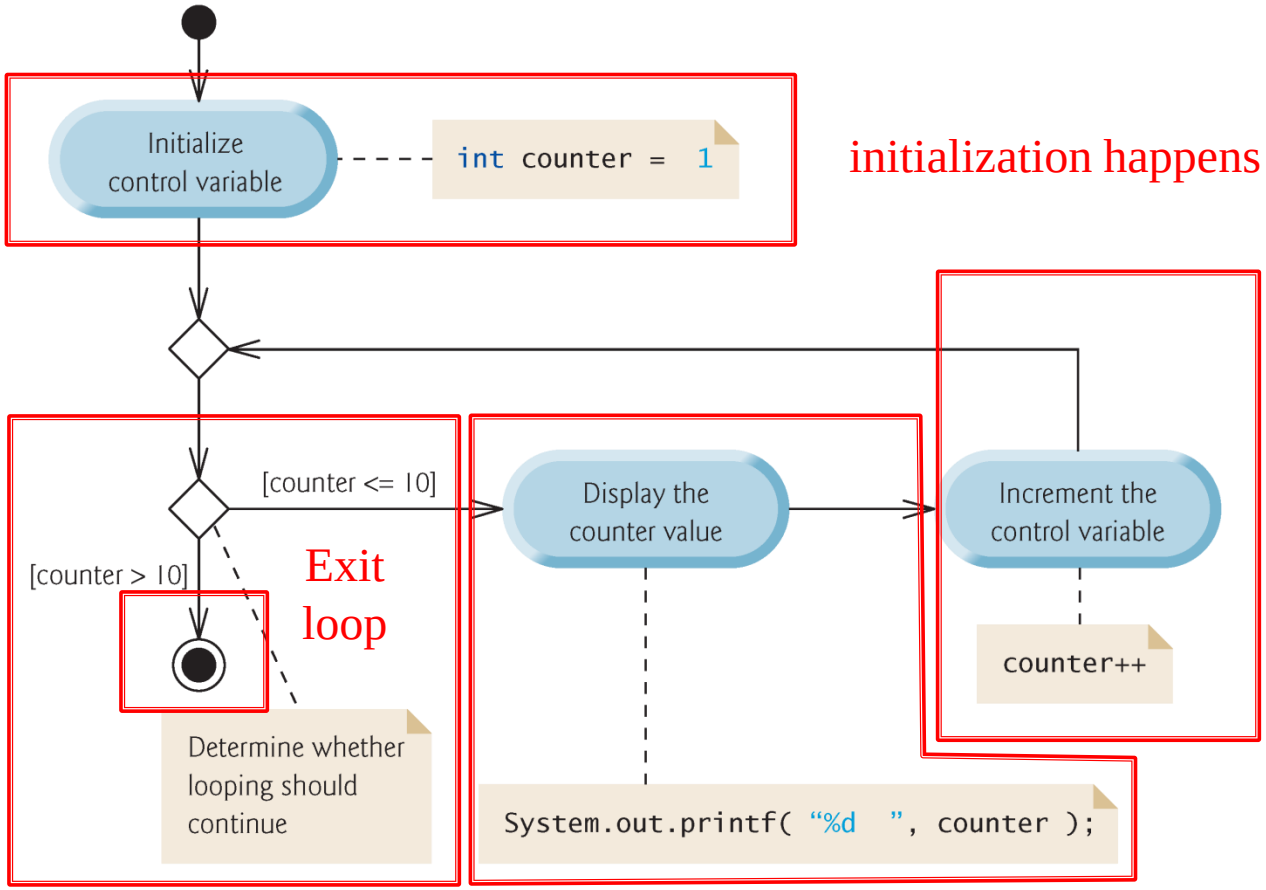
- Specifies the counter-controlled-repetition details in a single line of code

```
public class ForCounter {

    public static void main(String[] args) {

        for(int counter = 1; counter <= 10; counter++) {

            System.out.printf("%d", counter);

        }

        System.out.println();

    }

}
```

The for Repetition Statement



Execution Flow of for Loop



initialization happens first and only one time

Counter increment and then back to condition evaluation

Condition evaluation on each iteration

Loop body execution

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

```
for(initialization; loop-continuation condition; increment/decrement exp) {
    statement(s);
}
```

```
initialization;
while(loop-continuation condition) {
   statement(s);
   increment/decrement exp;
}
```

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 (definition): storing a value in a variable.
Initialization is the first assignment.

```
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.

```
for(int i = 1; ; i++){
    System.out.println("indefinite loop");
}
```

- You might omit the *initialization* expression if the program initializes the control variable before the loop.

```
int i = 0;
for(; i <= 10; i++){
     System.out.println(i);
}
```

More on for Repetition Statement

- You might omit the *increment* if the program calculates it with statements in the loop's body or no increment is needed.

```
for(int i = 1; i <= 10; ){
    System.out.println("i");
    i ++;
}
```

More on for Repetition Statement

- 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* expressions can contain multiple expressions separated by commas （逗号）.

```
for ( int i = 2;  i <= 20; total += i, i += 2 ) {
    System.out.println(total);
}
```

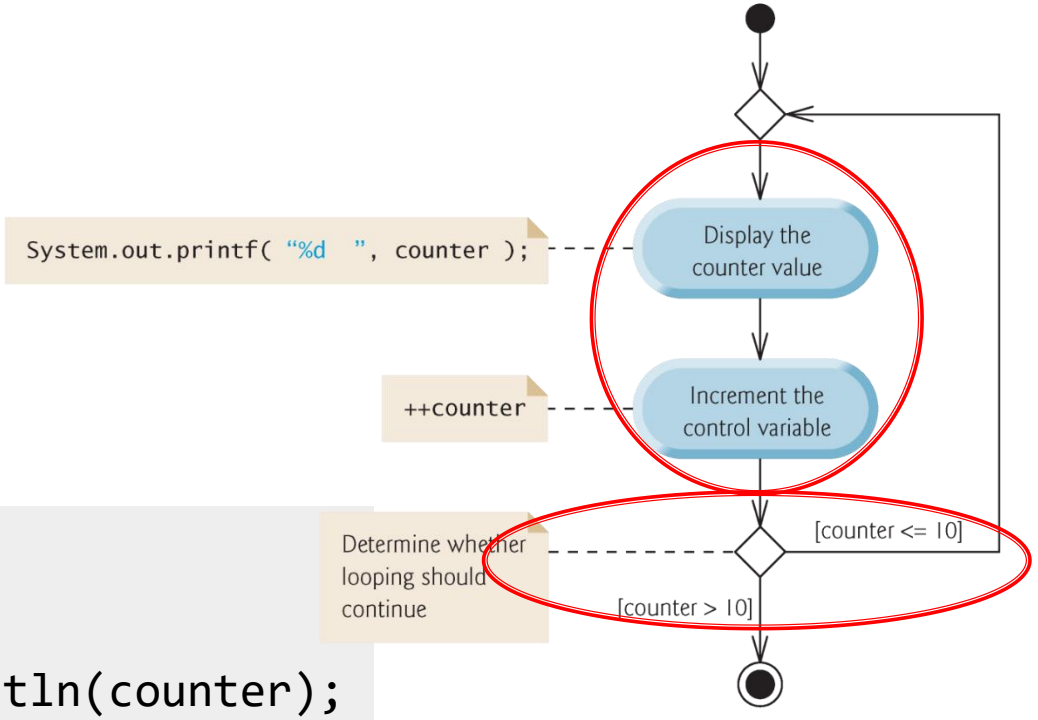
Equivalent to:

```
for ( int i = 2; i <= 20; i += 2 ) {
    System.out.println(total);
    total += i;     // why not before println
}
```

The do…while repetition statement

- **do…while** is similar to **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




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

Execute loop body and then test

Don't forget semicolon

The `switch` Multiple-Selection Statement

```
if ( studentGrade == 'A' )
   System.out.println( "90-100" );
else if ( studentGrade == 'B' )
   System.out.println( "80-89" );
else if ( studentGrade == 'C' )
   System.out.println( "70-79" );
else if ( studentGrade == 'D' )
   System.out.println( "60-69" );
else
   System.out.println( "Score < 60" );
```

- Letter grade to score range

The `switch` Multiple-Selection Statement

```
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`* statement performs different actions based on the values of a **constant integral expression 整型常量表达式** 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.**

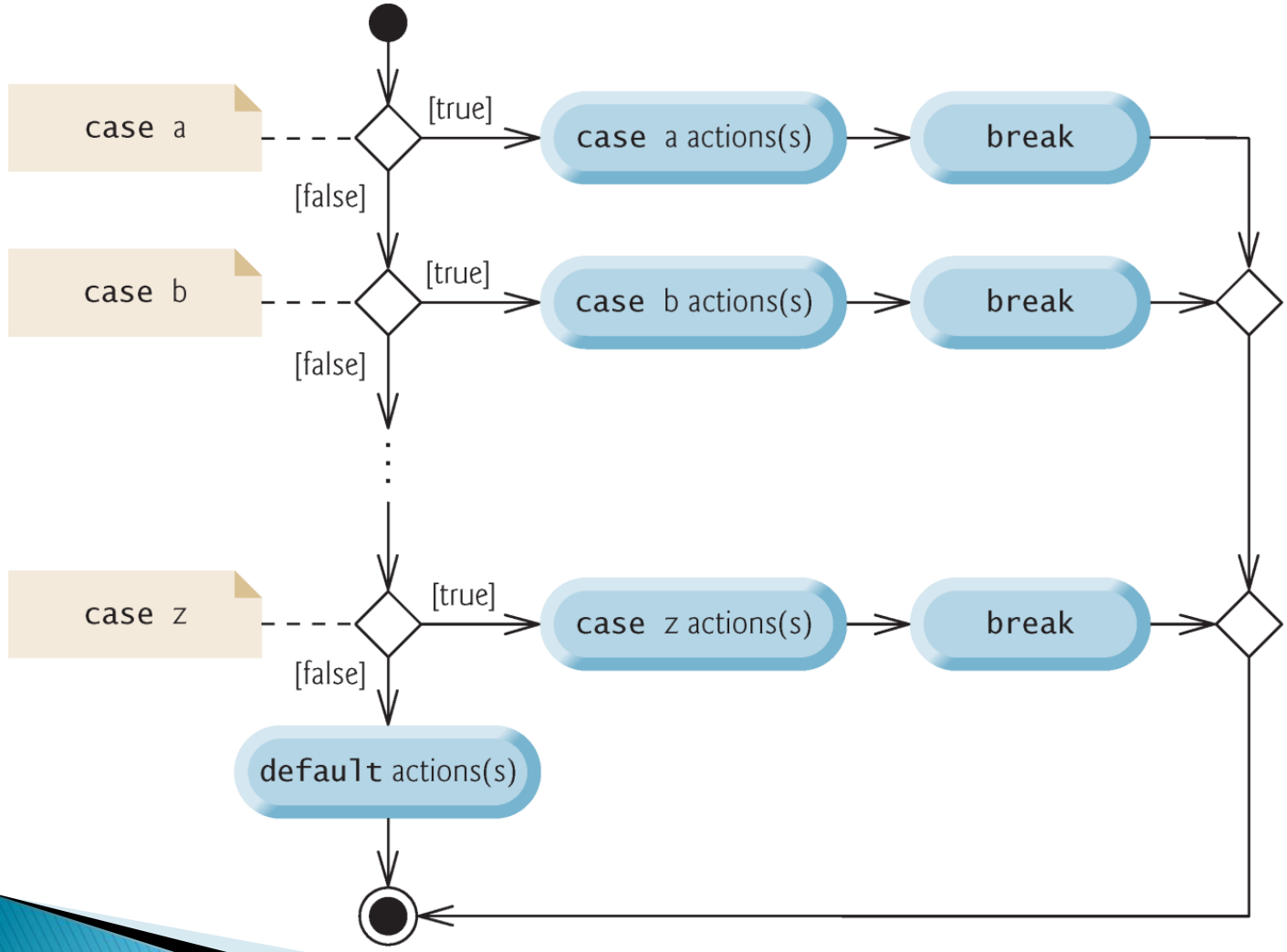
The `switch` Multiple-Selection Statement

controlling expression

```
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 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`.

Execution flow of switch



The switch Multiple-Selection Statement

```
switch (studentGrade) {
    case 90 <= studentGrade:
        System.out.println("A Level");
        break;
    case …:…
}
```

- `switch` does not provide a mechanism for testing ranges of values—every value must be listed in a separate case label.

```
switch (studentGrade) {
    case 'A' : {
        System.out.println("90 - 100");
        break;
    }
    case …:…
}
```

- Each case can have multiple statements (braces are optional)

The switch Multiple-Selection Statement

```
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");
}
```

- **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

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`.

```
public class BreakTest {
 // main method begins execution of Java application
    public static void main(String[] args) {
        int count;

        for (count = 1; count <= 10; count ++){
            if (count == 5)  // if count is 5,
                break;       // terminate loop
            System.out.printf ( "%d ", count );
        }  // end for

        System.out.printf( "\nLoop terminates at count = %d\n", count );
    } // end method main
}
```

```
1 2 3 4
Terminate the 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.

```
public class ContinueTest {
 // main method begins execution of Java application
    public static void main(String[] args) {
        int count;

        for (count = 1; count <= 10; count ++){
            if (count == 5)     // if count is 5,
                continue;       // skip the remaining of the loop
            System.out.printf ( "%d ", count );
        }  // end for

        System.out.printf( "\nLoop terminates at count = %d\n", count );
    } // end method main
}
```

```
1 2 3 4 6 7 8 9 10
Loop terminates at count = 11
```

Logical Operators （逻辑操作符）

- **Logical operators** help form complex conditions by combining simple ones:
  - && (conditional AND) 与
  - || (conditional OR) 或
  - & (boolean logical AND) 按位与
  - | (boolean logical inclusive OR)按位或
  - ^ (boolean logical exclusive OR) 按位异或
  - ! (logical NOT) 非
- &, | 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

```
a && b || c
```

Evaluate first
(&& has higher precedence)

```
a || b || c
```

Evaluate first
(|| is left associative)

Short-circuit evaluation of && and || （短路求值）

- 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 )` `|| ( 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.

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

Prints 0

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

Prints 6

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

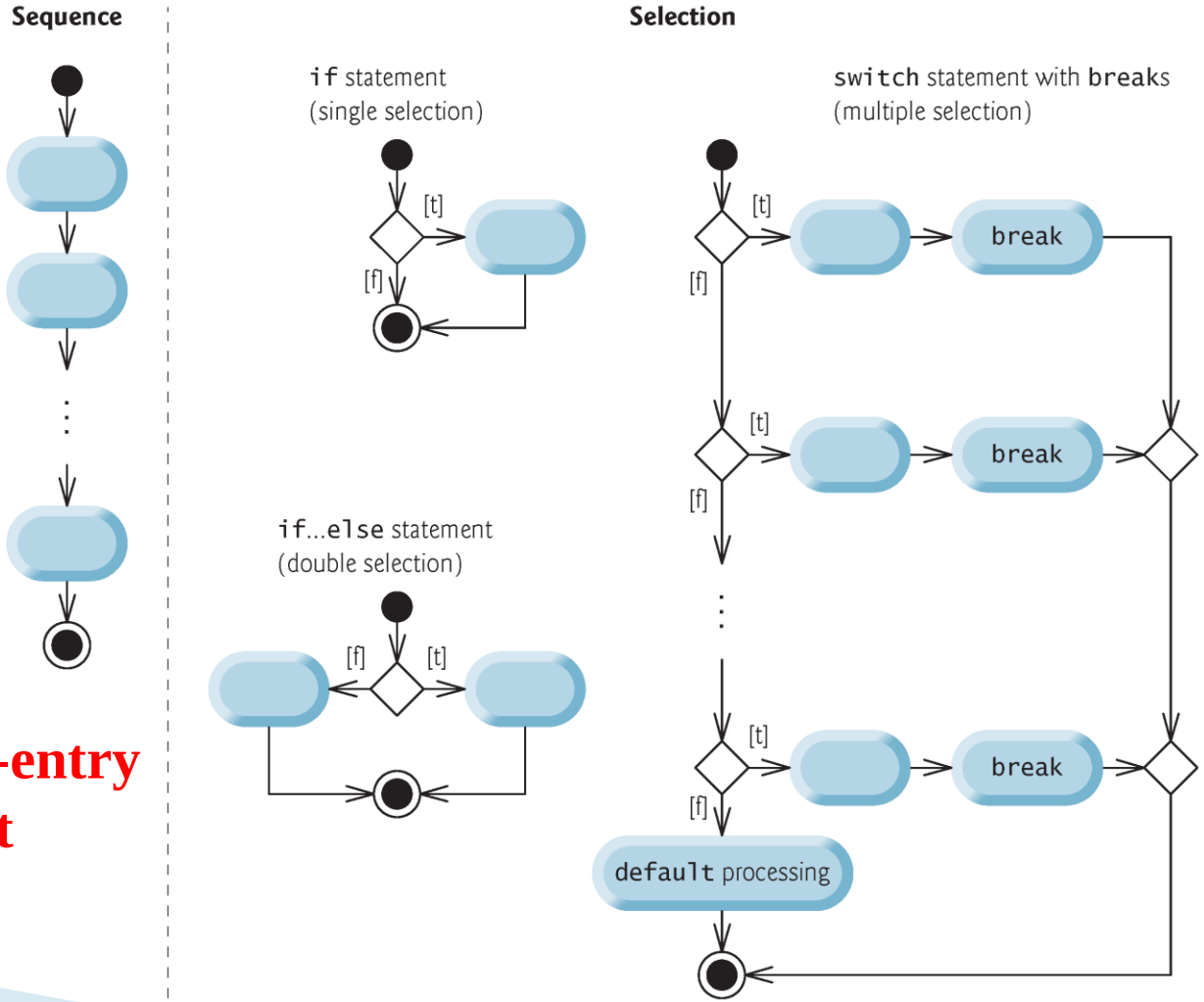
The Operators Introduced So Far

Precedence

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

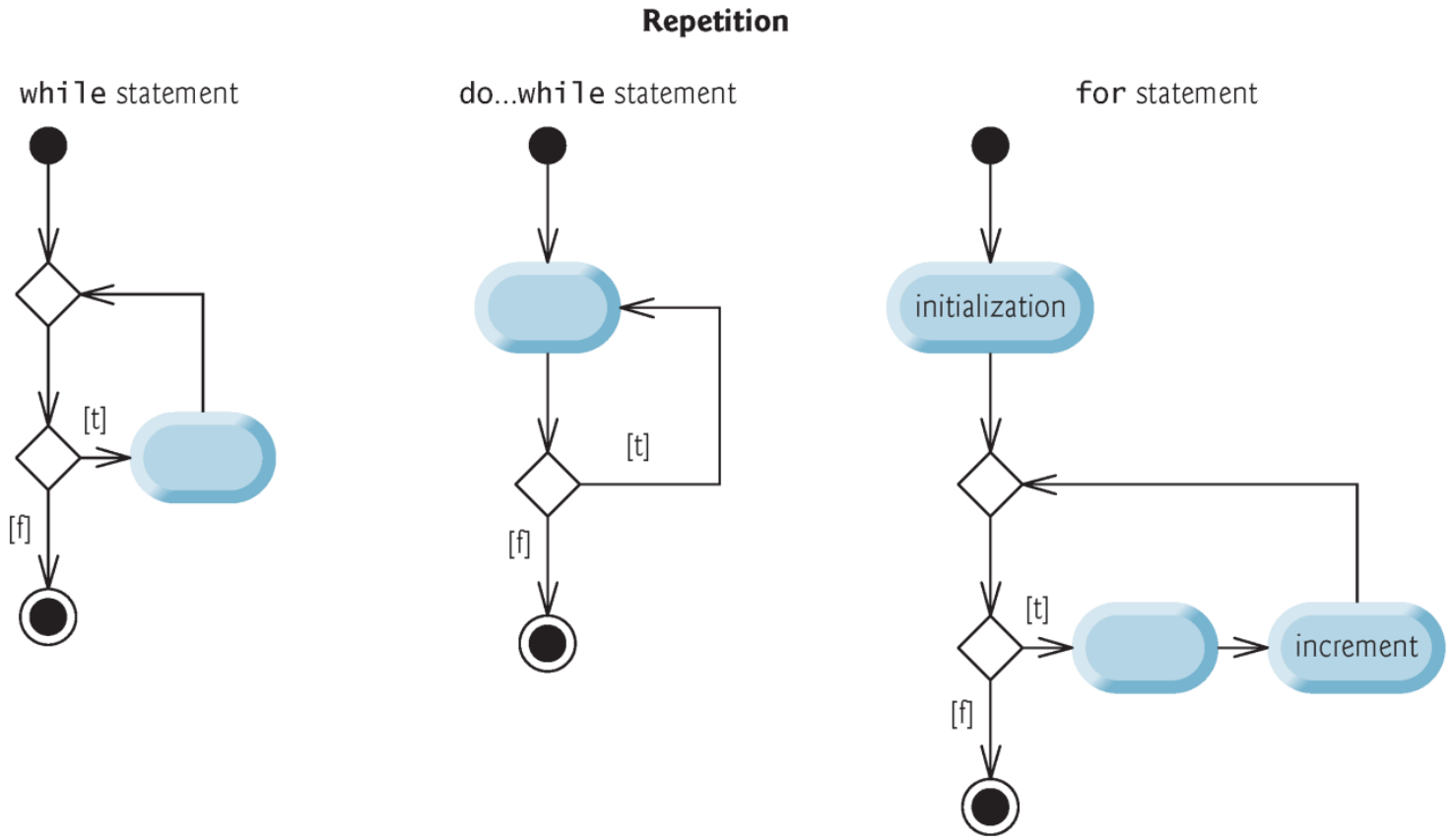
Fig. 4.18 | Precedence/associativity of the operators discussed so far.

Control Structures Summary



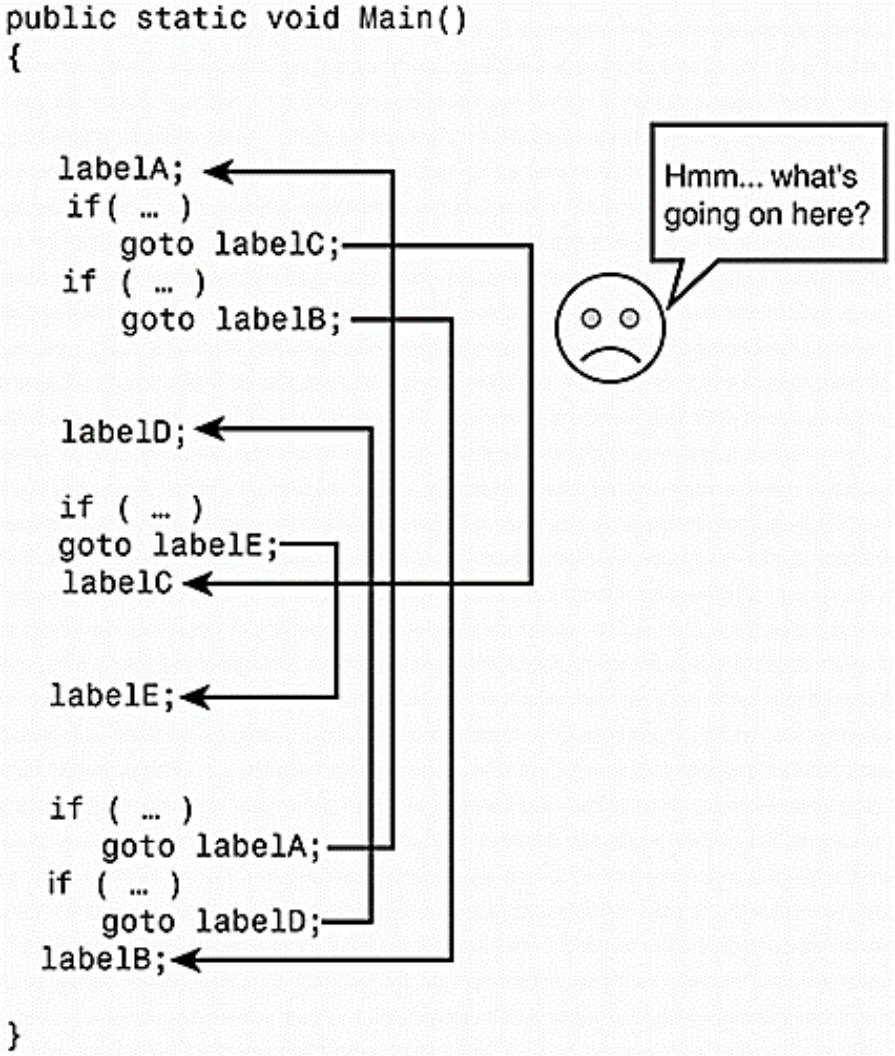
Always single-entry and single-exit

Control Structures Summary



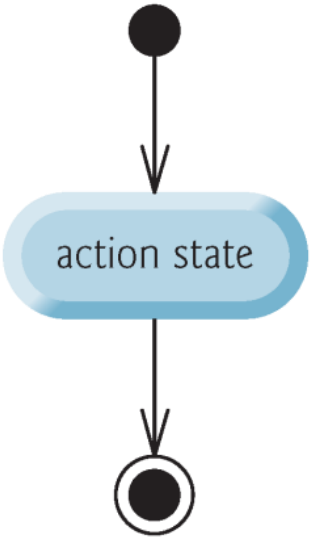
Structured Programming （结构化编程）

Structured programming makes extensive use of controls structures to produce programs with high quality and clarity (in contrast to using simple jumps such as the goto statement).



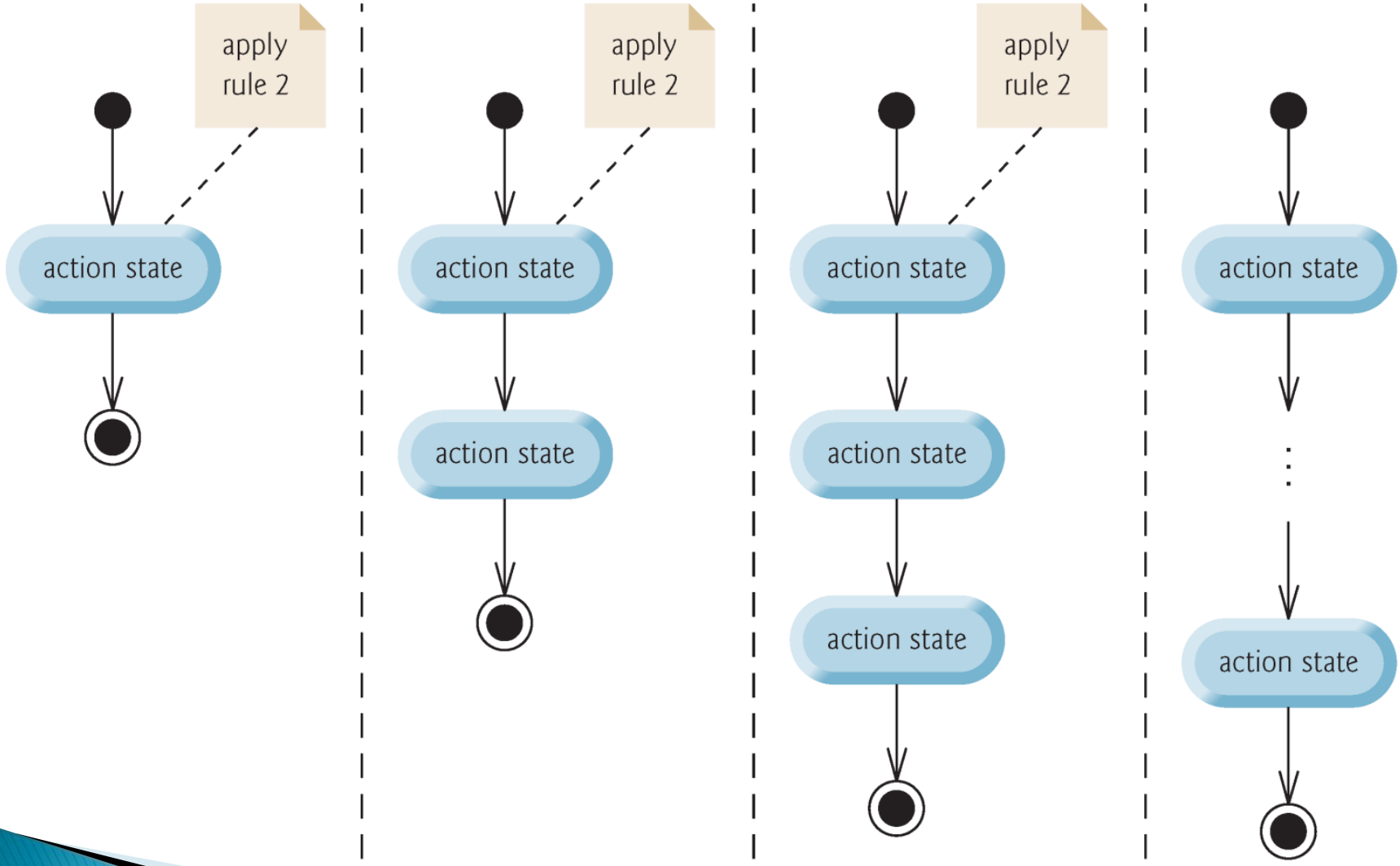
Rules for Forming Structured Programs

- Begin with the simplest activity diagram.
- **Stacking Rule （堆叠规则）：** Any action state can be replaced by two action states in sequence.
- **Nesting Rule （嵌套规则）：** Any action state can be replaced by any control statement (sequence of action states, `if`, `if…else`, `switch`, `while`, `do…while` or `for`).
- Stacking rule and nesting rule can be applied as often as you like and in any order.

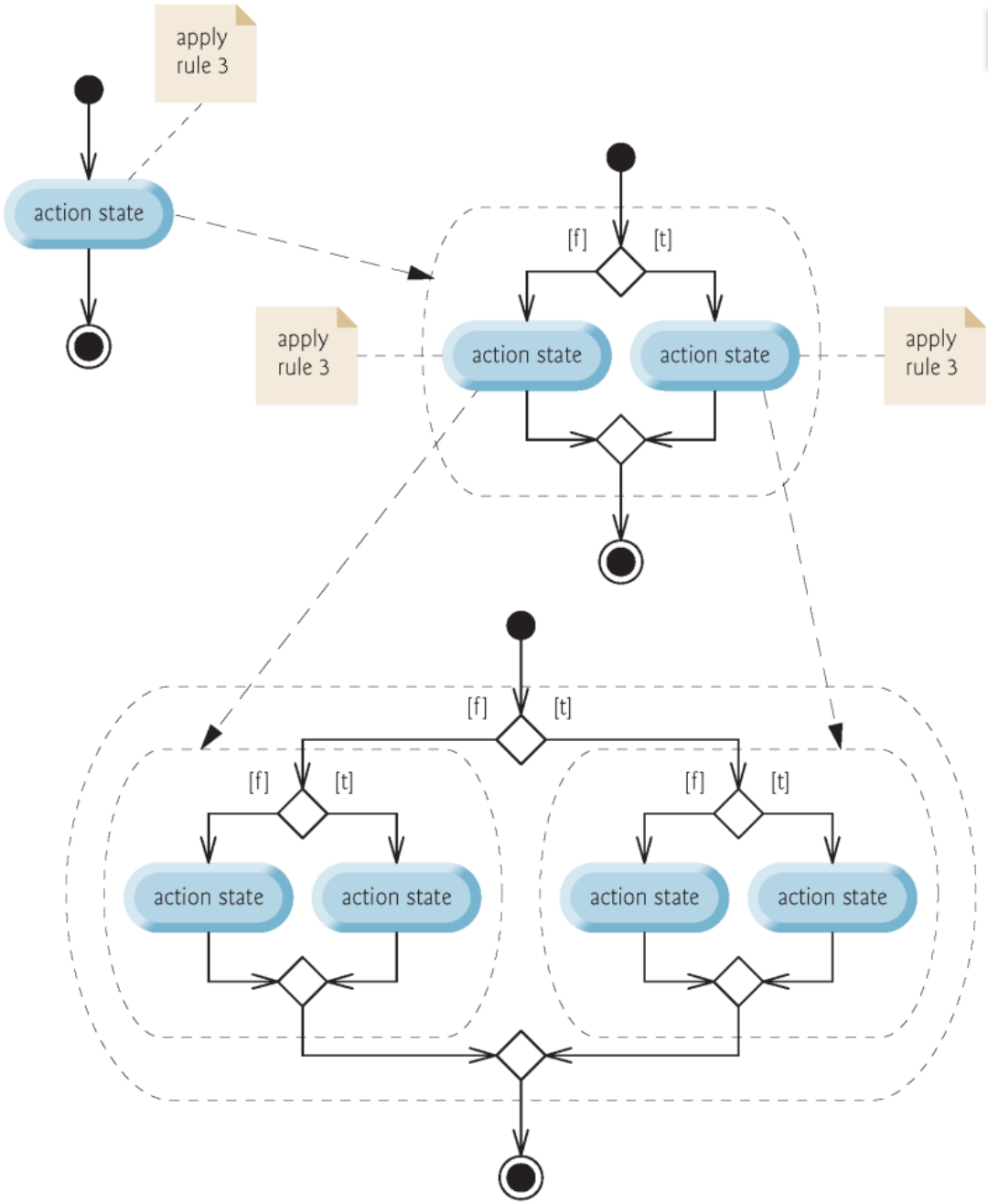


Begin with the simplest activity diagram.

Apply stacking rule



Apply nesting rule



Structured Programming Summary

- Bohm and Jacopini （伯姆-贾可皮尼定理）: Only three forms of control are needed to implement any algorithm:
  - Sequence
  - Selection
  - Repetition
- The sequence structure is trivial.

Structured Programming Summary

- Selection is implemented in one of three ways:
  - `if` statement (single selection)
  - `if…else` statement (double selection)
  - `switch` statement (multiple selection)
- The simple `if` statement is sufficient to provide any form of selection—everything that can be done with the `if…else` and `switch` can be implemented by combining `if` statements.

Structured Programming Summary

- Repetition is implemented in one of three ways:
  - `while` statement
  - `do…while` statement
  - `for` statement
- The `while` statement is sufficient to provide any form of repetition. Everything that can be done with `do…while` and `for` can be done with the `while` statement.

Structured Programming Summary

- In essence, any form of control ever needed in a Java program can be expressed in terms of
  - sequence
  - `if` statement (selection)
  - `while` statement (repetition)

  and that these can be combined in only two ways—stacking and nesting.

Assignment 1

- Submit on OJ !!!
- OJ instructions on Sakai
- Deadline：2021.10.17

Declaration form

- Submit on Sakai
  - Print and sign
  - Digital signation

- Deadline： 2021.10.12