



Chapter 3

Control Statements (Part II)

TAO Yida

taoyd@sustech.edu.cn

Objectives

- ▶ To use **for** and **do...while** statements
- ▶ To use **switch** statement
- ▶ To use **continue** and **break** statements
- ▶ To use logical operators
- ▶ Structured programming

Counter-Controlled Repetition with `while`

```
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();  
    }  
}
```

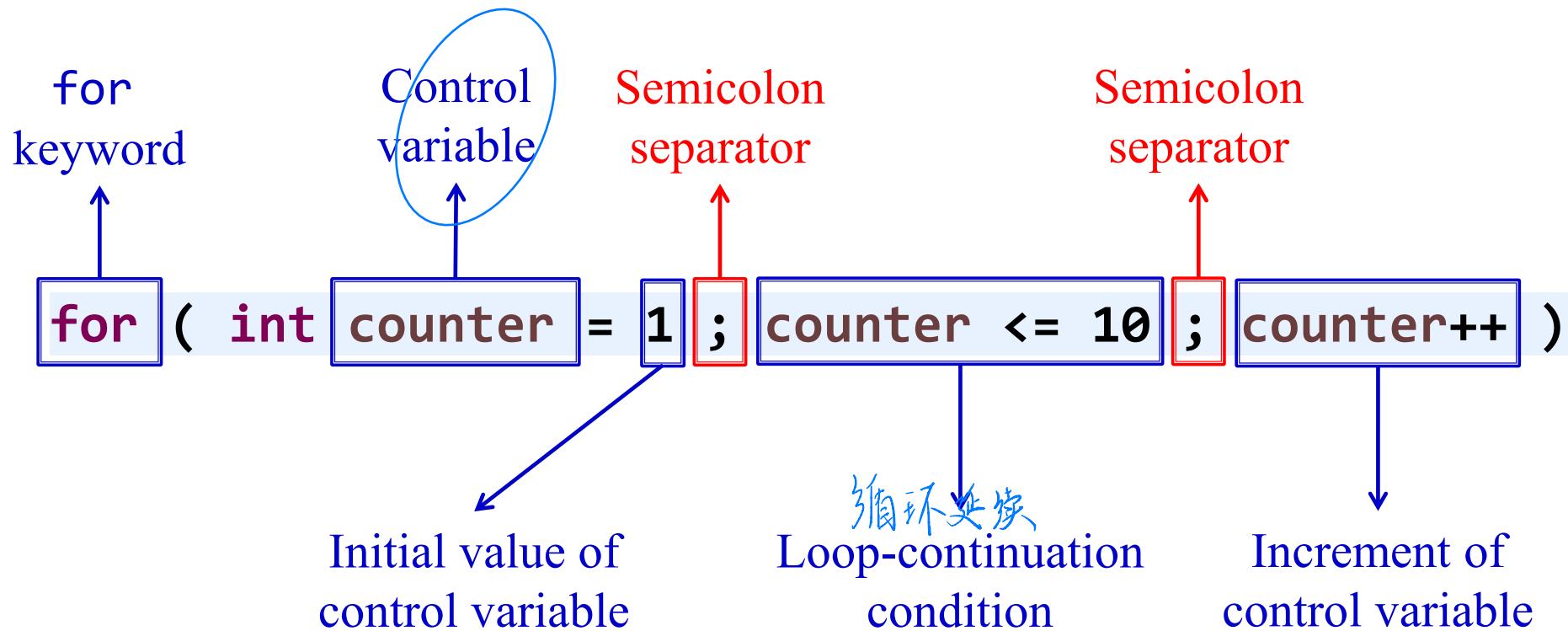


The for Repetition Statement

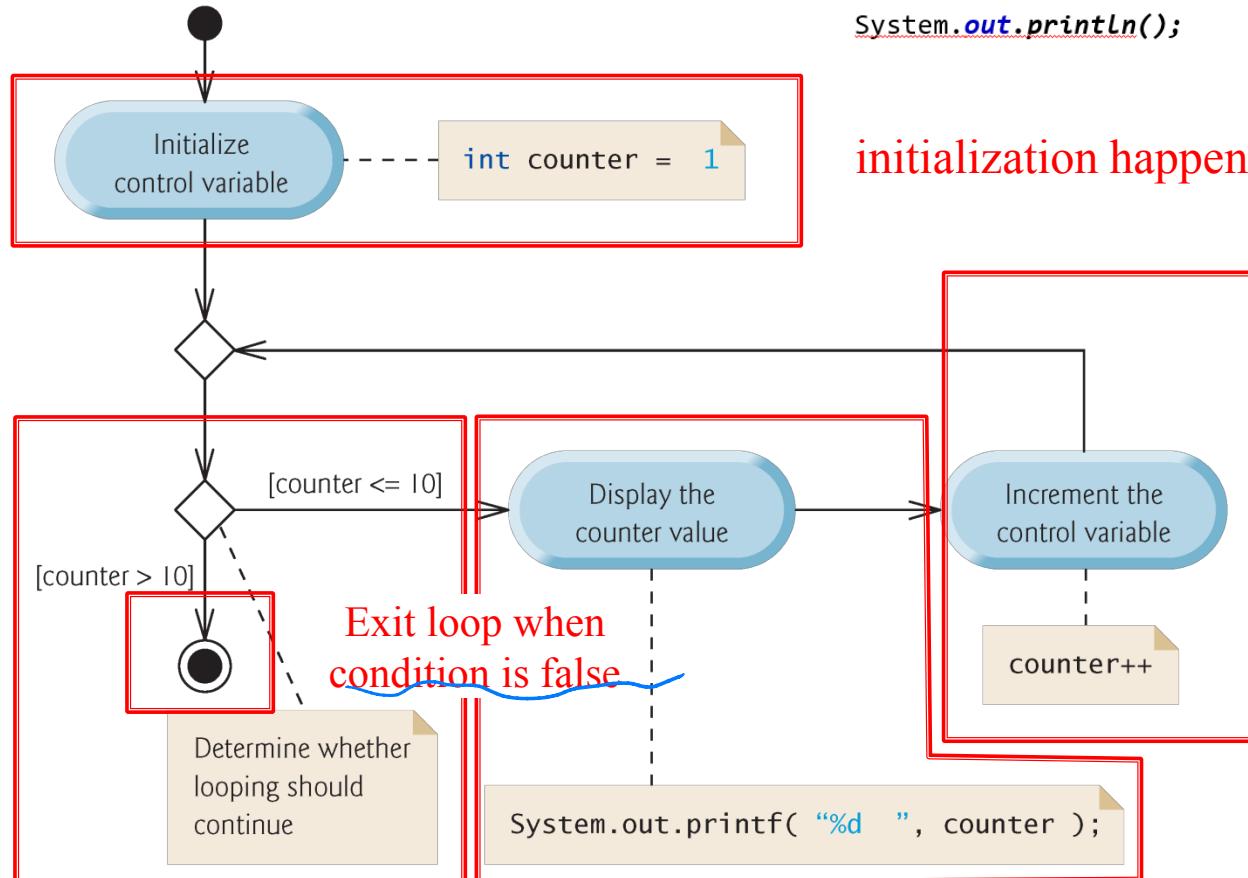
- ▶ Specifies the counter-controlled-repetition details in a single line of code

```
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



```

for(int counter = 1; counter <= 10; counter++) {
    System.out.printf("%d", counter);
}
System.out.println();

```

initialization happens first and only one time

Counter increment
and then back to
condition evaluation

Condition evaluation on each iteration

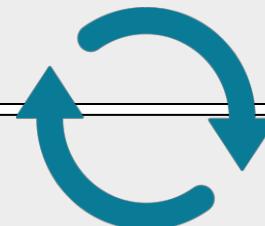
Execute loop body when condition is true

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

In most cases, a **for** statement can be easily represented with **an equivalent while statement**

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

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





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?  
}
```



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 = 0; ; i++) {  
    System.out.println("infinite loop");  
}
```

↑保留2个

- 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);  
}
```

=

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



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.

声明的位置 故名
声明的位

```
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

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

```
Scanner sc = new Scanner(System.in);  
int input = sc.nextInt();  
for( ; input > 0; ) {  
    System.out.println(input);  
    input = sc.nextInt();  
}  
sc.close();
```

控制输入值



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.

```

int total = 0;
for (int i = 2; i <= 10; total += i, i += 2) {
    System.out.println(total);
} // what's the output?

```

可以多条语句
②
再累加
先输出 ①
执行完循环体内先 total+=i 后 i+=2

=	i
0	2
2	4
6	6
12	8
20	10 打完 20
30	12 加成 但是 i > 10 不执行

```

int total = 0, i = 2;
while (i <= 10) {
    System.out.println(total);
    total += i;
    i += 2;
}

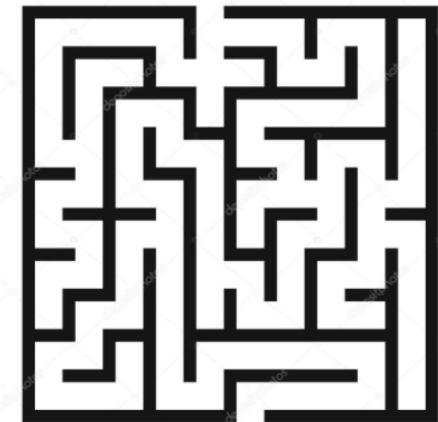
```

Using for or while loop?

- ▶ Typically, **for** statements are used for counter-controlled repetition and **while** statements for sentinel-controlled repetition
- 知道次数
不知次数
但有终止条件*

The required Reverse Pyramid pattern containing 8 rows is:

```
Row # 1 contains 8 stars : * * * * * * * *
Row # 2 contains 7 stars : * * * * * * *
Row # 3 contains 6 stars : * * * * * *
Row # 4 contains 5 stars : * * * * *
Row # 5 contains 4 stars : * * * *
Row # 6 contains 3 stars : * * *
Row # 7 contains 2 stars : * *
Row # 8 contains 1 stars : *
```



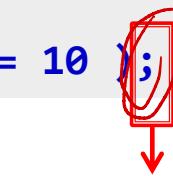
The do...while repetition statement

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

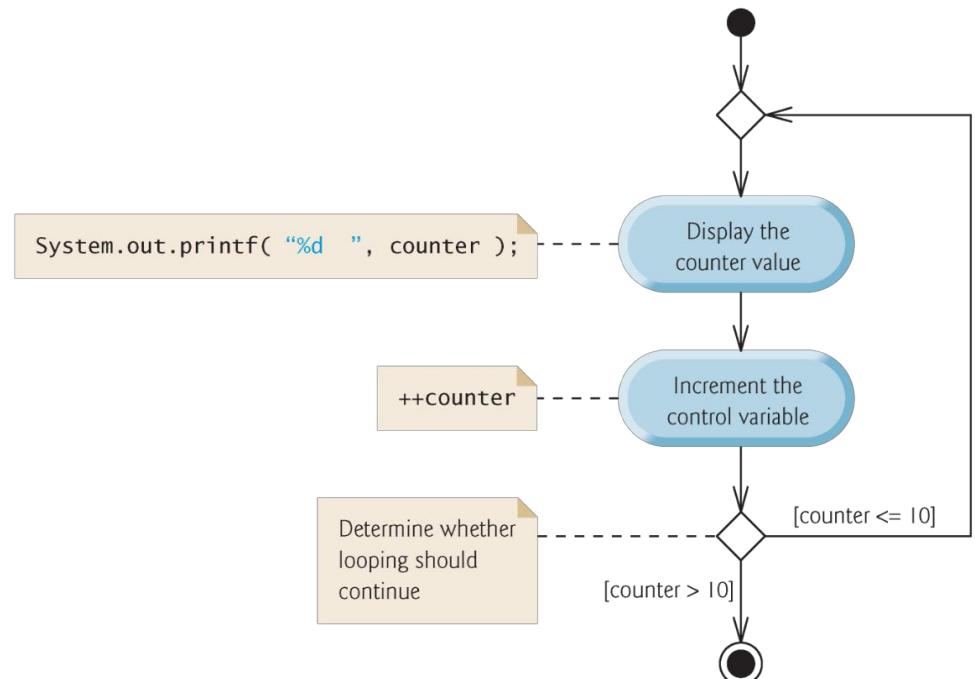
 第一次一定做 先做后判
最后一次相同

Execution flow of do...while

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



Don't forget
semicolon



Execute loop body and then test condition

do...while vs while

```
int num = 0;  
while(num>5){  
    System.out.println("num > 5");  
} No output
```

while: Condition is tested at the beginning of the loop

```
int num = 0;  
do{  
    System.out.println("num > 5");  
}while(num>5);
```

Output: num>5

do...while: Condition is tested at the end of the loop; body will be executed at least once

Objectives

- ▶ To use for and do...while statements
- ▶ To use **switch** statement
- ▶ To use **continue** and **break** statements
- ▶ To use logical operators
- ▶ Structured programming

Recall the if...else statement

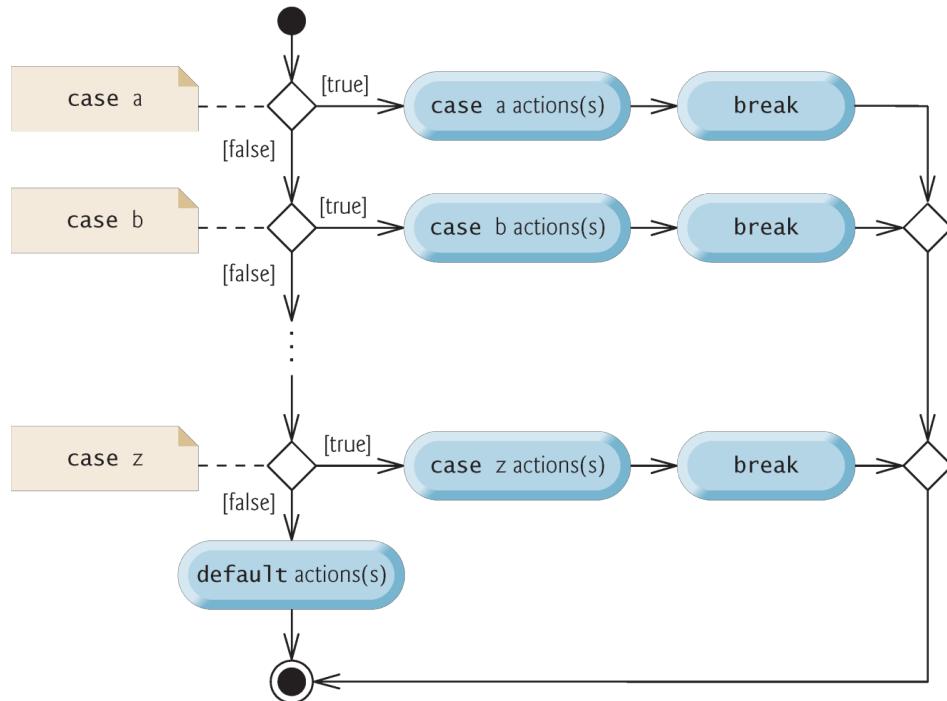
```
if(studentGrade == 'A') {  
    System.out.println("90 - 100");  
} else if(studentGrade == 'B') {  
    System.out.println("80 - 89");  
} else if(studentGrade == 'C') {  
    System.out.println("70 - 79");  
} else if(studentGrade == 'D') {  
    System.out.println("60 - 69");  
} else {  
   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

Letter grade



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 Multiple-Selection Statement

```
switch (switch-expression) {  
    case value1: statement(s)1;  
        break;  
    case value2: statement(s)2;  
        break;  
    ...  
    case valueN: statement(s)N;  
        break;  
    default: statement(s)-for-default;  
}
```

The switch-expression must yield a value of char, byte, short, int, or String type and must always be enclosed in parentheses.

```
switch (studentGrade) {  
    case 'A': 值  
        System.out.println("90 - 100");  
        break; 不能缺 看到第一个  
    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: 其他  
        System.out.println("score < 60");  
}
```



The switch Multiple-Selection Statement

```
switch (switch-expression) {  
    case value1: statement(s)1;  
        break;  
    case value2: statement(s)2;  
        break;  
    ...  
    case valueN: statement(s)N;  
        break;  
    default: statement(s)-for-default;  
}
```

- The value1, . . . , and valueN must have the same data type as the value of the switch expression.
- value1, . . . , and valueN are constant expressions, meaning that they cannot contain variables, such as $1 + x$

```
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 (grade) {  
    case 90 <= grade: X 不能是判断  
        System.out.println("A Level");  
        break;  
    case ...:...  
}
```

- The value1, . . . , and valueN must have the **same data type** as the value of the switch expression.
- value1, . . . , and valueN are **constant expressions**, meaning that they **cannot contain variables**, such as $1 + x$

```
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 (switch-expression) {  
    case value1: statement(s)1;  
        break;  
    case value2: statement(s)2;  
        break;  
    ...  
    case valueN: statement(s)N;  
        break;  
    default: statement(s)-for-default;  
}
```

When the value in a case statement **matches** the value of the switch-expression, the statements **starting from this case** are executed until either a **break** statement or **the end of the switch statement** is reached.

The keyword **break** is optional. The break statement **immediately ends the switch 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");  
}
```

其
他
defa
ult



The switch Multiple-Selection Statement

```
switch (switch-expression) {  
    case value1: statement(s)1;  
        break;  
    case value2: statement(s)2;  
        break;  
    ...  
    case valueN: statement(s)N;  
        break;  
    default: statement(s)-for-default;  
}
```

The **default** case, which is optional, can be used to perform actions when **none of the specified cases matches the switch-expression**.

If no match occurs and there is no default case, program simply continues with the first statement after switch.

```
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");  
}
```

Using break in switch

```
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.

没有 break 执行下一个

If `studentGrade == 'A'`, then output is

90 – 100
80 – 89
70 – 79

Using break in switch

```
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
60 – 69
score < 60

Using break in switch

```
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 == 'C'`, then output is

70 – 79
60 – 69
score < 60

Using break in switch

```
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.

To avoid programming errors and improve code maintainability, it is a good idea to put a comment in a case clause if `break` is purposely omitted.

相同的故意不写 `break`



switch vs if...else

- ▶ if...else
 - Can test expressions based on ranges of values or conditions;
Better for conditions that result into a boolean
- ▶ switch
 - Better for fixed data values, e.g., int, char, String

```
if ( studentGrade >= 90 )
    System.out.println( "A" );
else if ( studentGrade >= 80 )
    System.out.println( "B" );
else if ( studentGrade >= 70 )
    System.out.println( "C" );
else if ( studentGrade >= 60 )
    System.out.println( "D" );
else
    System.out.println( "F" );
```

```
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");
}
```

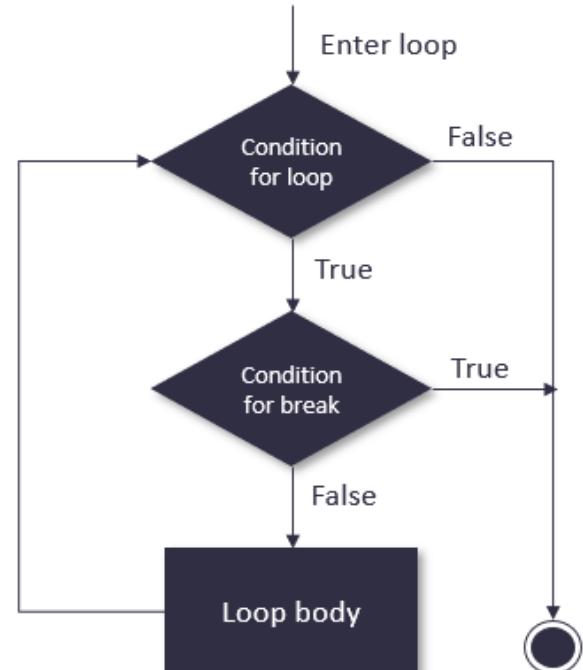


Objectives

- ▶ To use for and do...while statements
- ▶ To use switch statement
- ▶ To use **continue** and **break** statements
- ▶ To use logical operators
- ▶ Structured programming

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**.



break: jump out of the loop



The break Statement

```
public class BreakTest {  
    public static void main(String[] args) {  
        int count;  
        for(count = 1; count <= 10; count++) { // loop 10 times  
            if(count == 5) {  
                break; // terminate loop if count == 5  
            }  
            System.out.printf("%d ", count);  
        }  
        System.out.printf("\nBroke out of loop at count = %d\n", count);  
    }  
}
```

i 由停在5不动

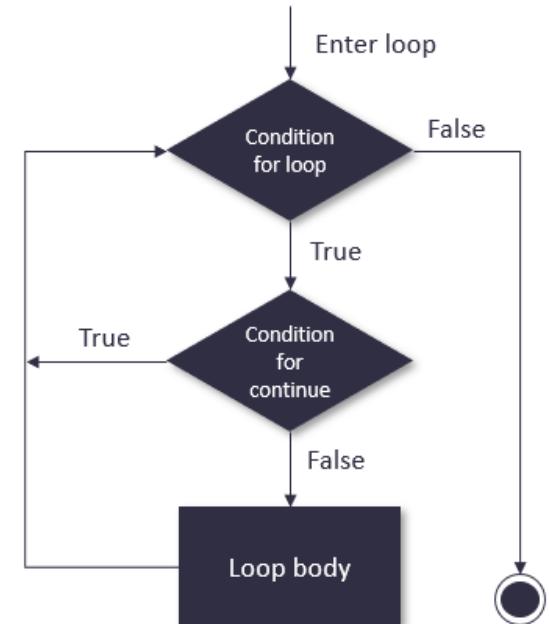
先判断再是否输出

1 2 3 4 0

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.



continue: skip one iteration if a condition is satisfied, then continue with the next iteration



The continue Statement

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

执行下一个循环

当前循环不再执行

1 2 3 4 6 7 8 9 10

Used continue to skip printing 5

~~0..!~~
while to for 不互换得例。
continue / for 跳过执行
while 跳过先执行判断



Objectives

- ▶ To use for and do...while statements
- ▶ To use switch statement
- ▶ To use continue and break statements
- ▶ To use logical operators (逻辑运算符)
- ▶ Structured programming

Logical Operators

- The logical operators `!`, `&&`, `||`, and `^` can be used to create a compound Boolean expression.

TABLE 3.3 Boolean Operators

<i>Operator</i>	<i>Name</i>	<i>Description</i>
<code>!</code>	not	logical negation
<code>&&</code>	and	logical conjunction
<code> </code>	or	logical disjunction
<code>^</code>	exclusive or	logical exclusion

The ! Operator

- ▶ **!** (also known as **logical negation** or **logical complement**)
unary operator “reverses” the value of a condition.

TABLE 3.4 Truth Table for Operator !

p	$\neg p$	Example (assume <code>age = 24, weight = 140</code>)
<code>true</code>	<code>false</code>	$\neg(\text{age} > 18)$ is <code>false</code> , because $(\text{age} > 18)$ is <code>true</code> .
<code>false</code>	<code>true</code>	一元运算符 “ <code>==</code> ” $\neg(\text{weight} == 150)$ is <code>true</code> , because $(\text{weight} == 150)$ is <code>false</code> .

The && Operator

- ▶ **&&** ensures that two conditions on its left- and right-hand sides are *both true* before choosing a certain path of execution.

TABLE 3.5 Truth Table for Operator &&

p ₁	p ₂	p ₁ && p ₂	Example (assume age = 24, weight = 140)
false	false	false	
false	true	false	(age > 28) && (weight <= 140) is false , because (age > 28) is false .
true	false	false	
true	true	true	(age > 18) && (weight >= 140) is true , because (age > 18) and (weight >= 140) are both true .

The || Operator

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

TABLE 3.6 Truth Table for Operator `||`

<code>p₁</code>	<code>p₂</code>	<code>p₁ p₂</code>	<i>Example (assume <code>age = 24, weight = 140</code>)</i>
<code>false</code>	<code>false</code>	<code>false</code>	<code>(age > 34) (weight >= 150)</code> is <code>false</code> , because <code>(age > 34)</code> and <code>(weight >= 150)</code> are both <code>false</code> .
<code>false</code>	<code>true</code>	<code>true</code>	
<code>true</code>	<code>false</code>	<code>true</code>	<code>(age > 18) (weight < 140)</code> is <code>true</code> , because <code>(age > 18)</code> is <code>true</code> .
<code>true</code>	<code>true</code>	<code>true</code>	

The || Operator

- ▶ Operator `&&` has a higher precedence than operator `||`
- ▶ Both operators associate from left to right

$$\boxed{a \text{ } \&\& \text{ } b} \text{ } || \text{ } c$$

Evaluate first (precedence)
优先权

$$a \text{ } || \text{ } \boxed{b \text{ } || \text{ } c}$$

Evaluate first (associativity)
结合性



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.

若提前能判整体真值，后面可以不判断

- `(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*
- ▶ This is useful if the operand at the right-hand side of `&` or `|` has a required **side effect** (副作用)—a modification of a variable's value



Example: || vs. |

```
int b = 0, c = 0;  
if(true || b == (c = 6)) {  
    System.out.println(c); // what's c's value?  
}
```

Prints 0

```
int b = 0, c = 0;  
if(true | b == (c = 6)) {  
    System.out.println(c); // what's c's value?  
}
```

Prints 6

The ^ operator 异或

- 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**

不同为 true
相同 false

TABLE 3.7 Truth Table for Operator ^

p ₁	p ₂	p ₁ ^ p ₂	Example (assume age = 24, weight = 140)
false	false	false	(age > 34) ^ (weight > 140) is false , because (age > 34) and (weight > 140) are both false .
false	true	true	(age > 34) ^ (weight >= 140) is true , because (age > 34) is false but (weight >= 140) is true .
true	false	true	
true	true	false	

比特运算符

Bitwise Operators

- ▶ &, | and ^ are also bitwise operators when applied to integral operands.

→ 转二进制

a = 5 = 0101 (In Binary)
b = 7 = 0111 (In Binary)

Bitwise OR Operation of 5 and 7

0101
0111

0111 = 7 (In decimal)

操作

a = 5 = 0101 (In Binary)
b = 7 = 0111 (In Binary)

Bitwise AND Operation of 5 and 7

0101
& 0111

0101 = 5 (In decimal)

<https://www.geeksforgeeks.org/bitwise-operators-in-java/>



The Operators Introduced So Far

Precedence ↓

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

Associativity is not relevant for some operators.

For example, $(x \leq y) \leq z$ and $x++-$ and $++x++$ are **invalid** expressions in Java.

boolean 类型

不是所有结合律成立

不会法



General Rules

- ▶ The operators in expressions are evaluated in the order determined by the rules of parentheses, operator precedence, and operator associativity.
- ▶ Parentheses can be used to force the order of evaluation to occur in any sequence.
- ▶ Operators with higher precedence are evaluated earlier. For operators of the same precedence, their associativity determines the order of evaluation.
- ▶ ^{二元}All binary operators except assignment operators are left-associative; assignment operators are right-associative

When in doubt, use () or simply use multiple statements!

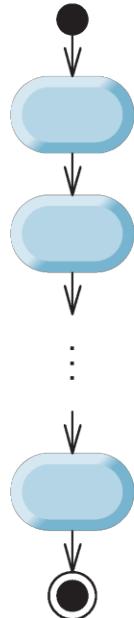


Objectives

- ▶ To use for and do...while statements
- ▶ To use switch statement
- ▶ To use continue and break statements
- ▶ To use logical operators (逻辑运算符)
- ▶ Structured programming

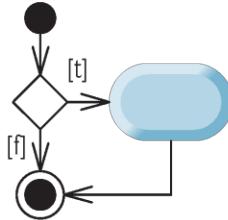
Control Structures Summary

Sequence

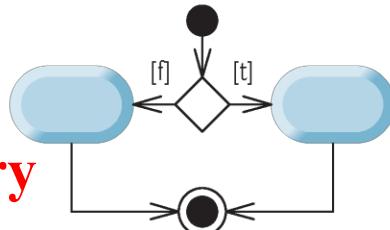


Selection

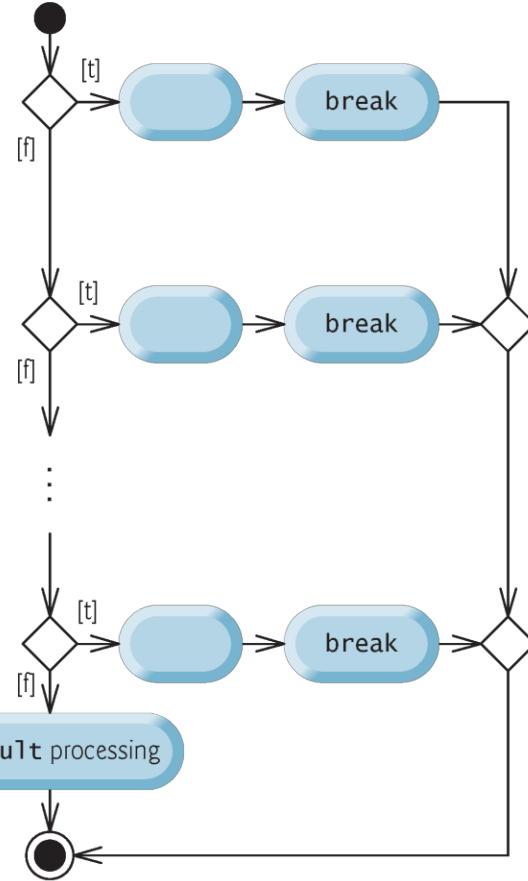
if statement
(single selection)



if...else statement
(double selection)



switch statement with **breaks**
(multiple selection)

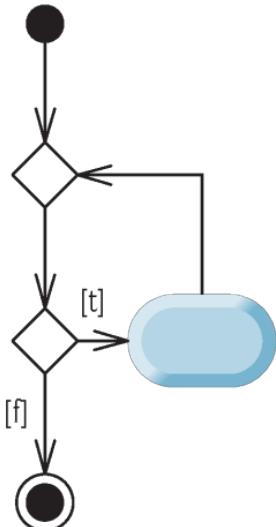


**Always single-entry
and single-exit**

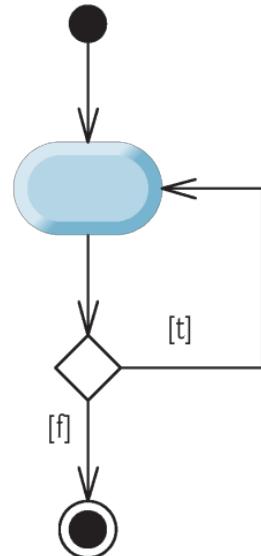
Control Structures Summary

Repetition

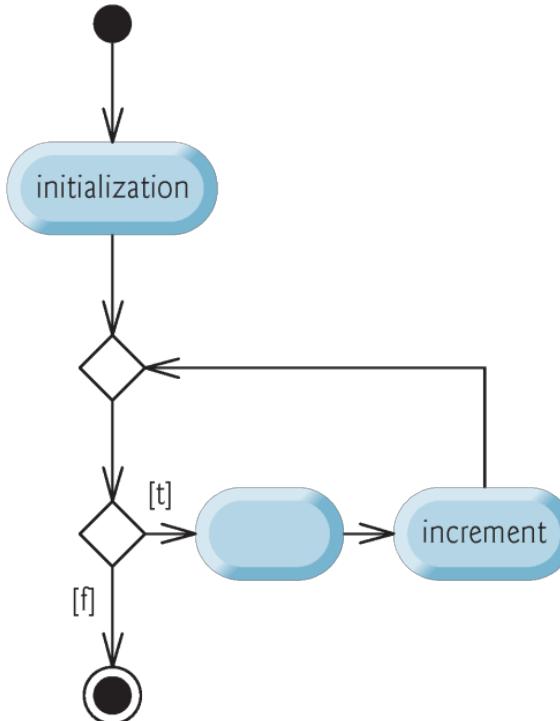
while statement



do...while statement



for statement



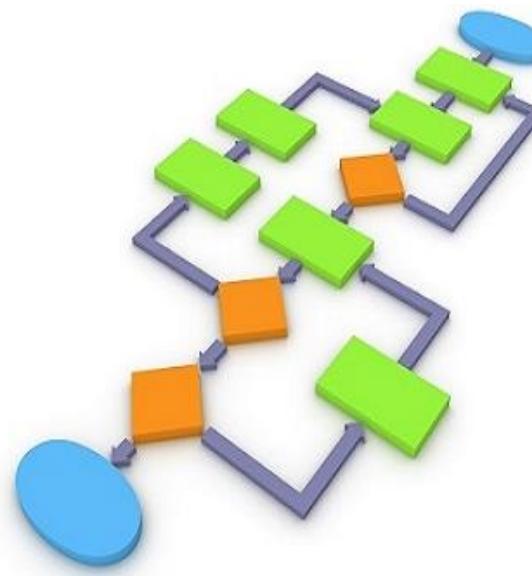
Structured Programming Summary

真理 + 规则 × 2

- ▶ **Böhm-Jacopini Theorem:** Only three forms of control are needed to implement any algorithm:

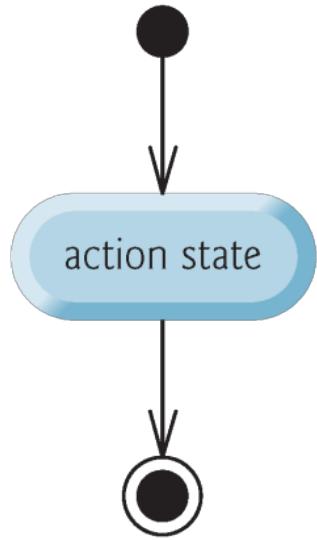
算法

- Sequence
- Selection
- Repetition



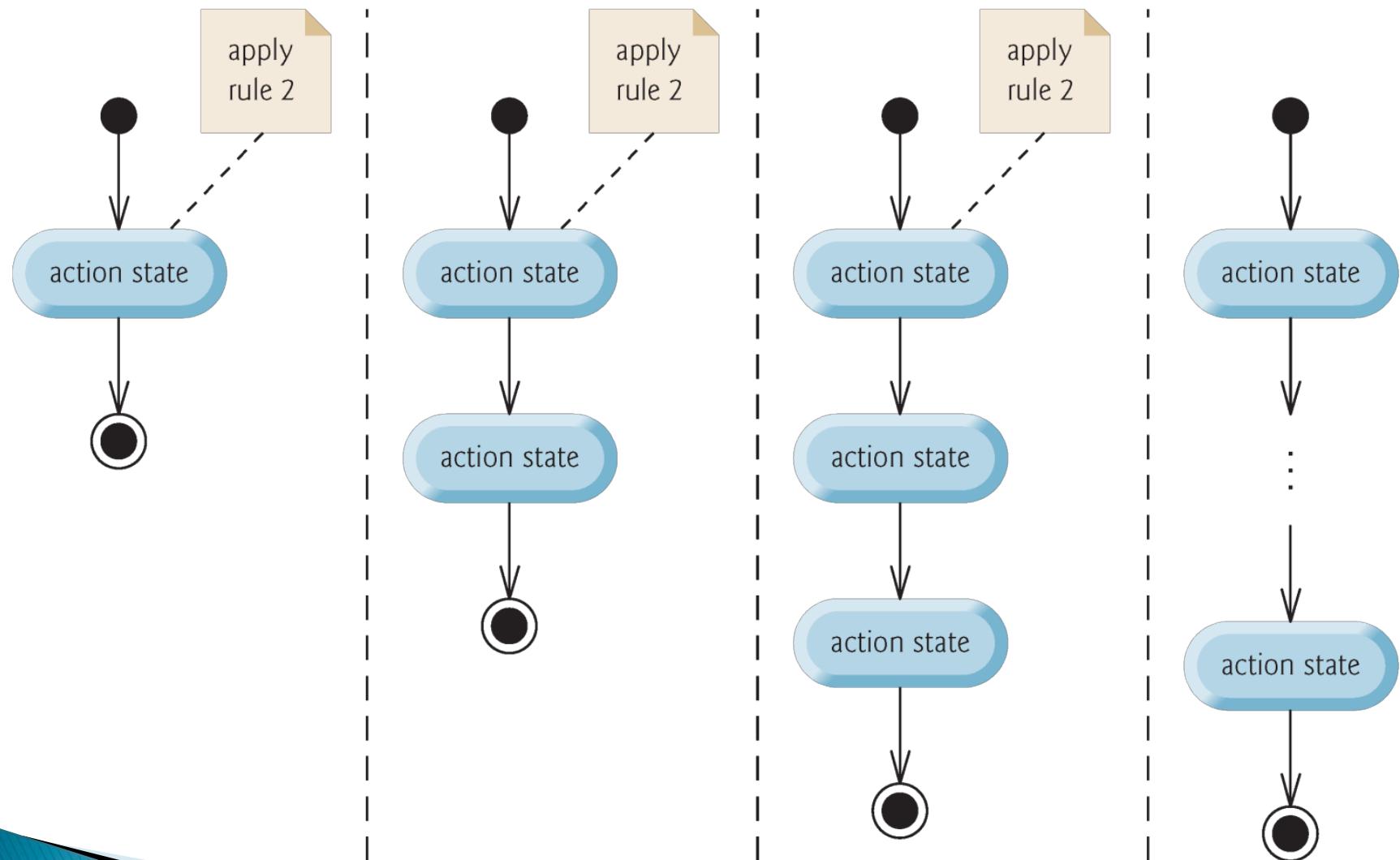
Rules for Forming Structured Programs

- ▶ 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**.

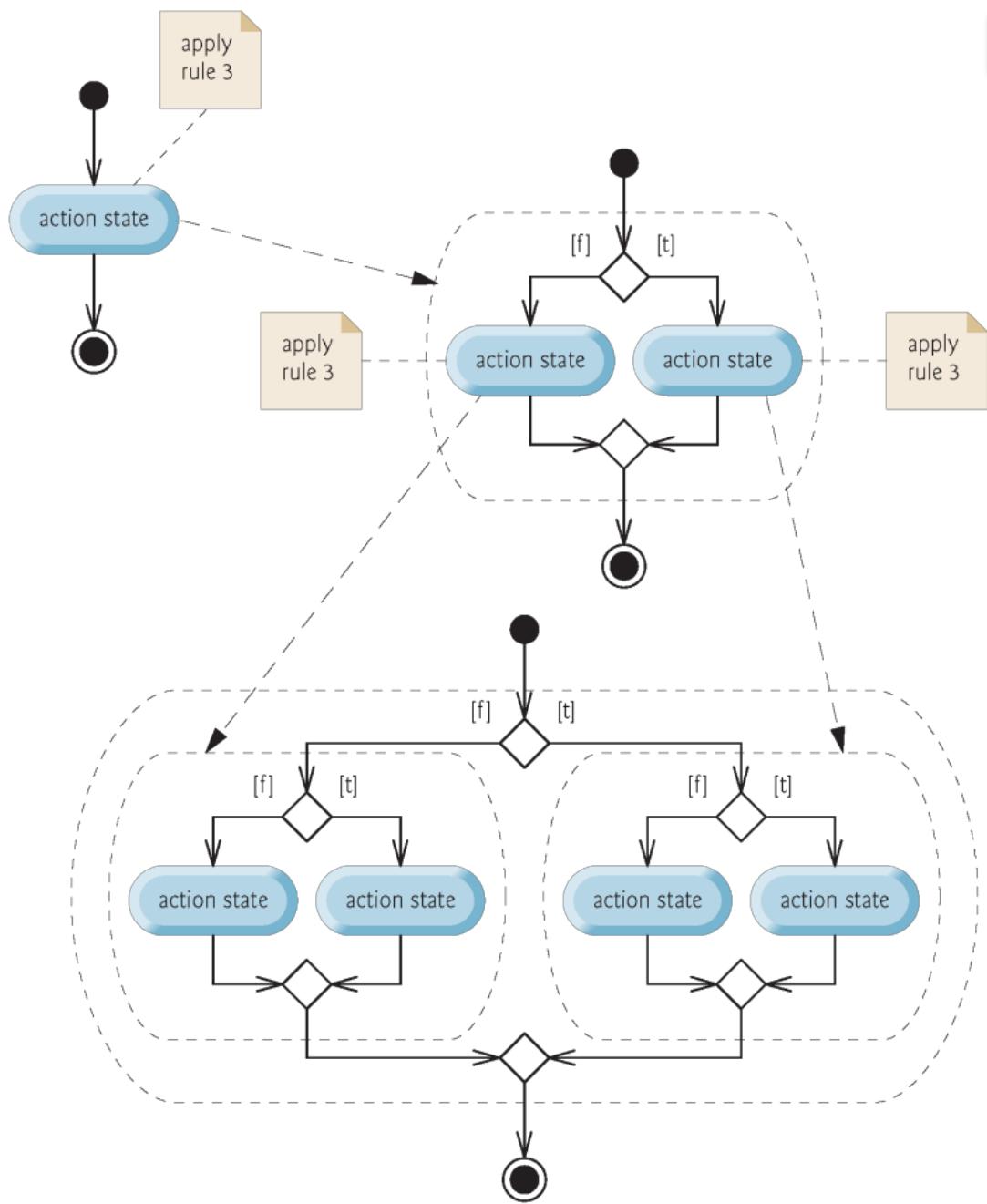


Begin with the simplest activity diagram.

Apply stacking rule



Apply nesting rule





Structured Programming Summary

- ▶ Selection is implemented in one of three ways:
 - `if` statement (single selection)
 - `if...else` statement (double selections)
 - `switch` statement (multiple selections)
- ▶ 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. 其他表达帮助简洁、方便

A Simple Case Study: Nested Loops

- ▶ Design a Java program to find all prime numbers (质数) within a user-specified range $[a, b]$

Algorithm formulation:

```
Get inputs a and b from users
For each integer c in [a, b]
    if c is a prime number
        print c
```



How to check?

Prime numbers can only be divided evenly by 1 and itself

A Simple Case Study: Nested Loops

- ▶ Design a Java program to find all prime numbers (质数) within a user-specified range $[a, b]$

Algorithm formulation:

```
Get inputs a and b from users  
For each integer c in [a, b]  
    if c is a prime number  
        print c
```

```
set isPrime to true  
For each integer d in [2, c-1]  
    if c % d is equal to 0  
        set isPrime to false  
        break
```



Java Code – Part 1

```
// in main method  
  
Scanner sc = new Scanner(System.in);  
System.out.print("Enter a number for a: ");  
int a = sc.nextInt();  
System.out.print("Enter a number for b: ");  
int b= sc.nextInt();  
if(a <= 1 || b < a) {  
    System.out.println("Invalid range!");  
    sc.close(); — 项目关闭节省资源。  
    return;  main方法存在  
}  
}
```

Java Code – Part 2

```
// a nested loop
for(int i = a; i <= b; i++) {
    boolean isPrime = true;

    for(int j = 2; j <= i - 1; j++) {
        if(i % j == 0) {
            isPrime = false;
            break;
        }
    }

    if(isPrime) {
        System.out.println(i);
    }
}

sc.close();
```

Inner loop

Outer loop