Chapter 4
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;

        while ( counter <= 10 ) {

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

            ++counter;

        }

        System.out.println();

    }

}
```

Control variable (loop counter)

Loop continuation condition

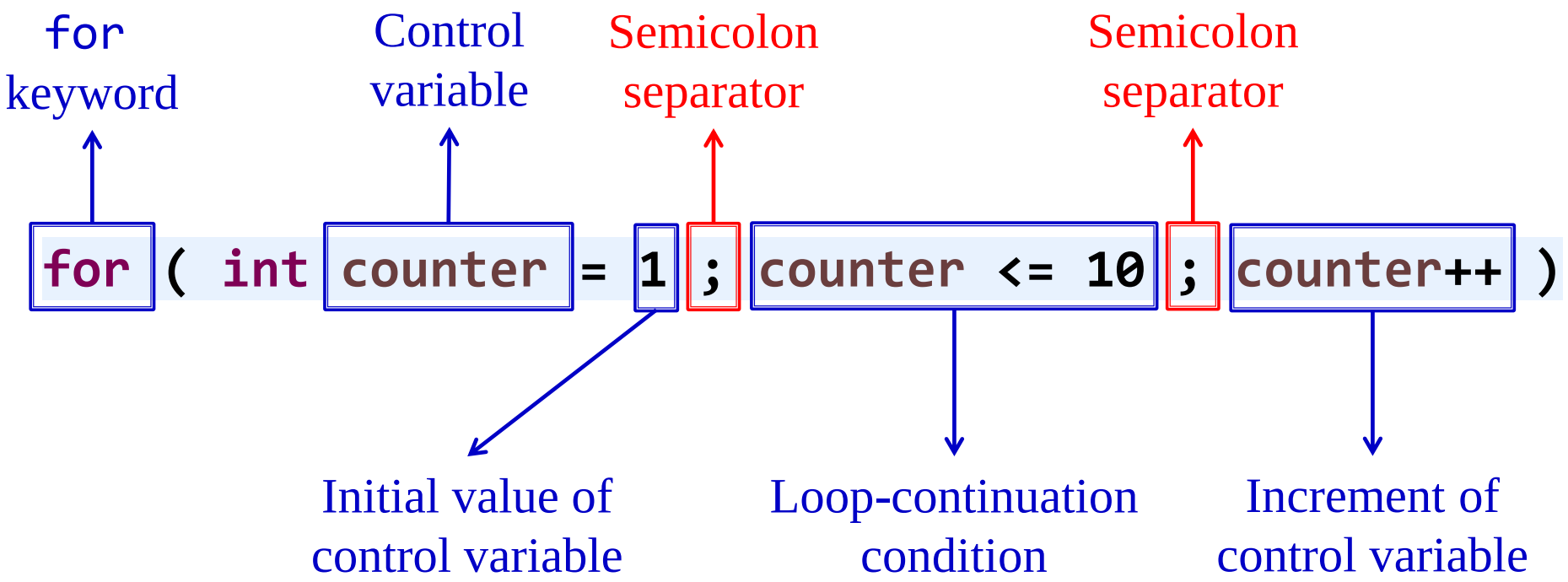
Counter increment (or decrement) in each iteration

The for Repetition Statement

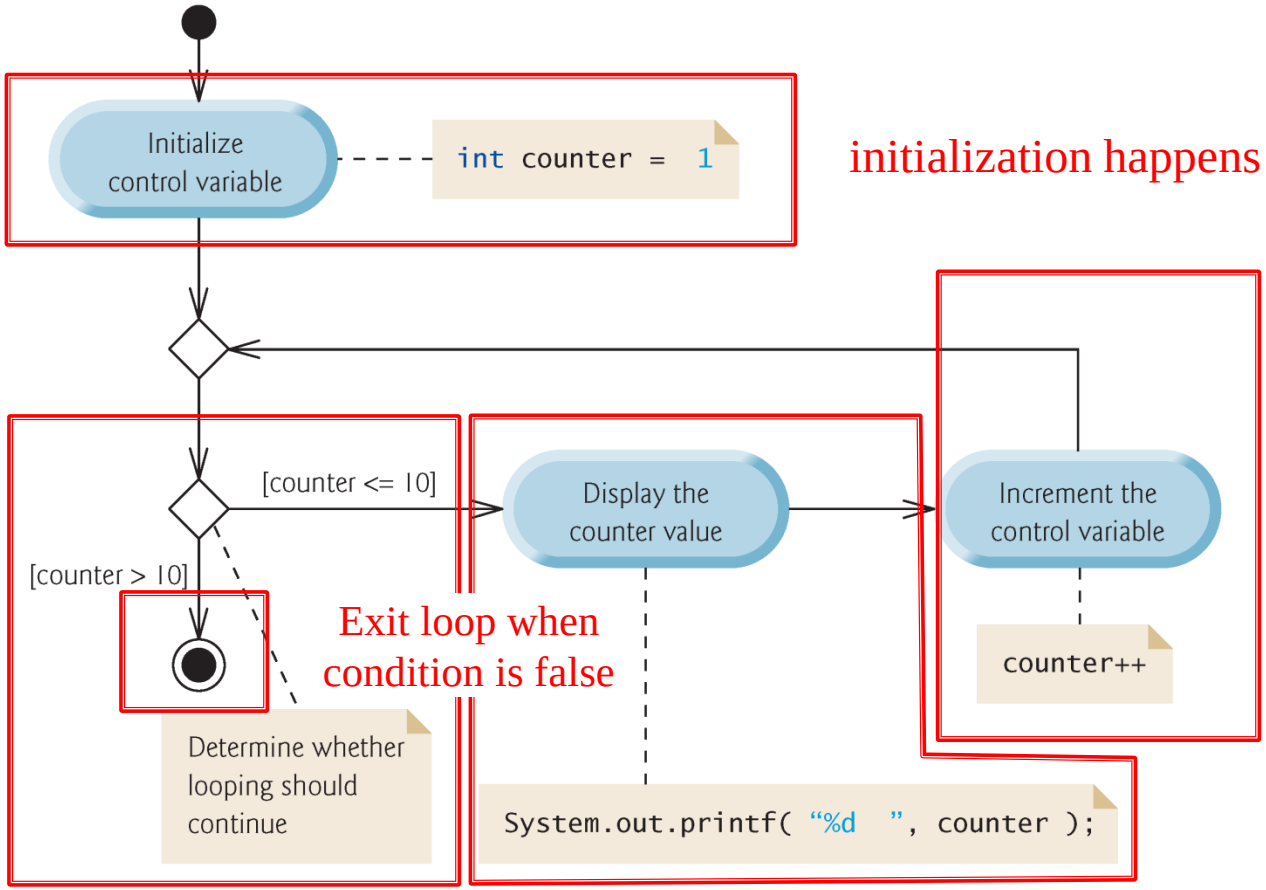
- Specifies the counter-controlled-repetition details in a single line of code

```
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

Execute loop body when condition is true

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");
}
```

- 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.

```
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();
```

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

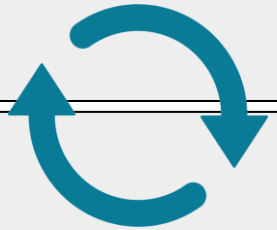
=

```
int total = 0;
for (int i = 2; i <= 20; i += 2) {
    System.out.println(total);
    total += i; // why last line?
}
```

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);
}
```

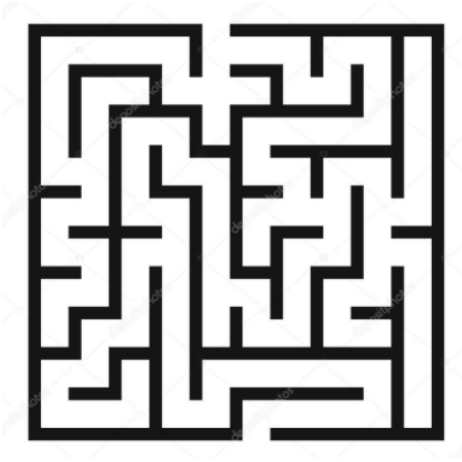


```
initialization;
while(loop-continuation condition) {
   statement(s);
   increment/decrement exp;
}
```

The `for` and `while` loops

- Typically, `for` statements are used for counter-controlled repetition and `while` statements for sentinel-controlled repetition

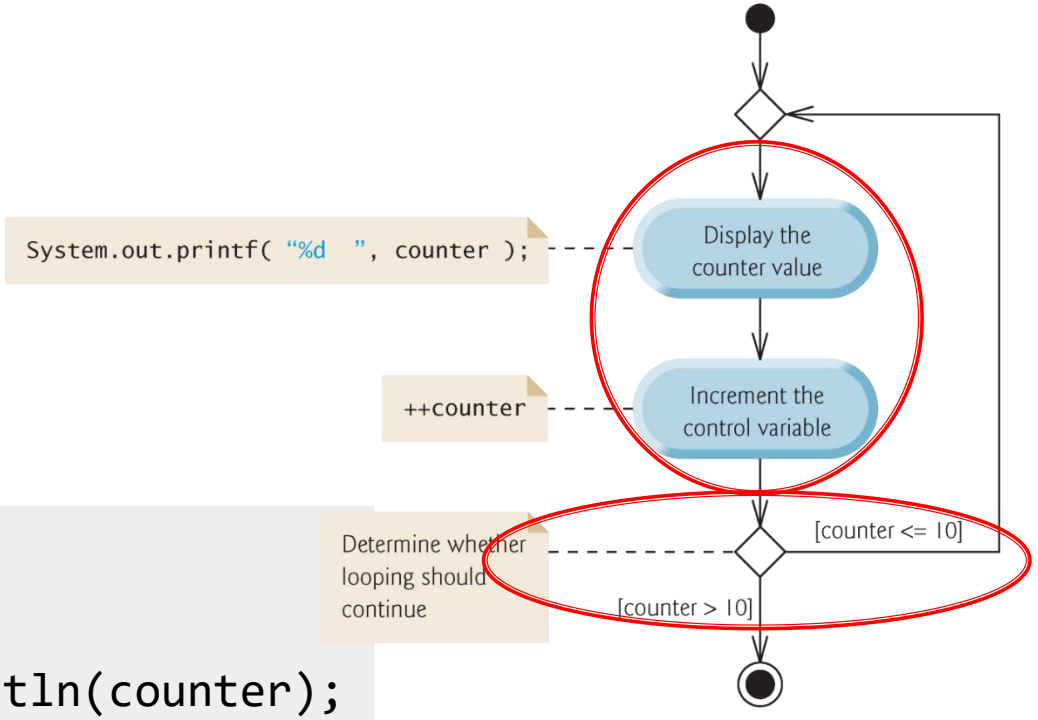
```
The required Reverse Pyramid pattern containing 8 rows is:

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.

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

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

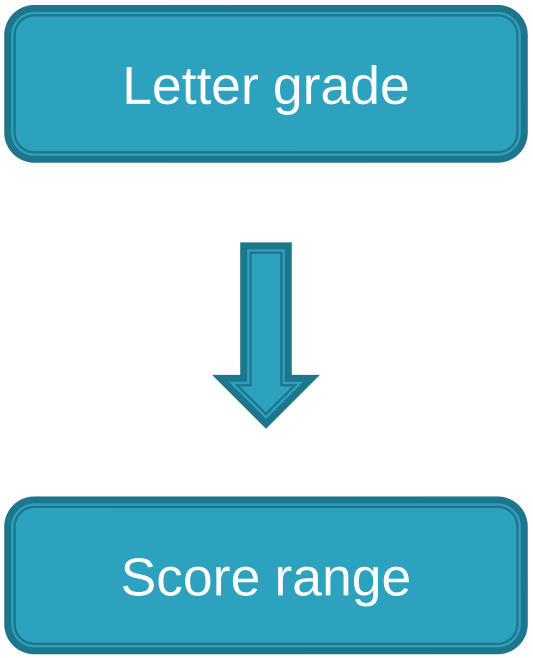
`do…while`: Condition is tested at the end of the loop; body will be executed at least once

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");
}
```



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 an **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.**

* Starting from Java 7, strings are also supported

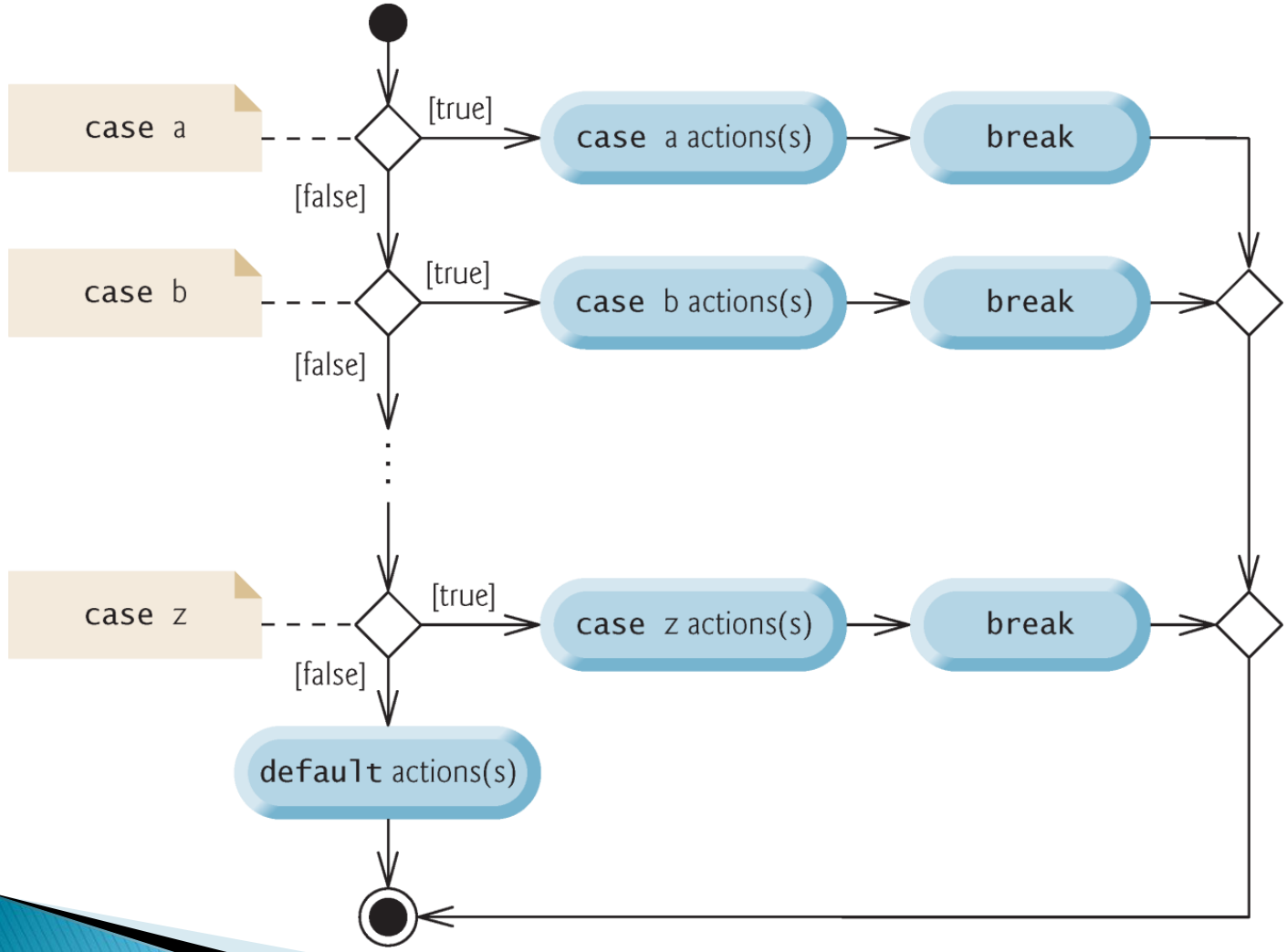
The `switch` Multiple-Selection Statement

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


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

- `switch` does not provide a mechanism for testing ranges of values—every value must be listed in a separate case label (if `grade` is `int`, case label should also be `int`)
- Each case can have multiple statements (braces are optional)

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");
}
```

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 `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
60 – 69
score < 60

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 == 'C'`, then output is

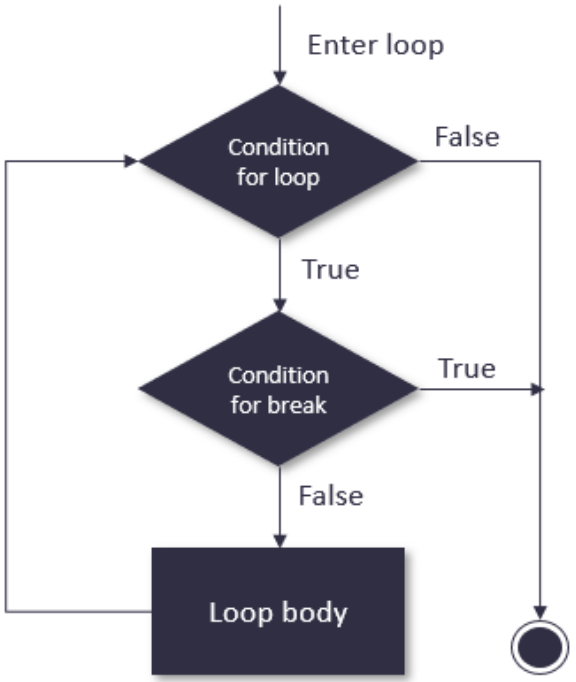
70 – 79
60 – 69
score < 60

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

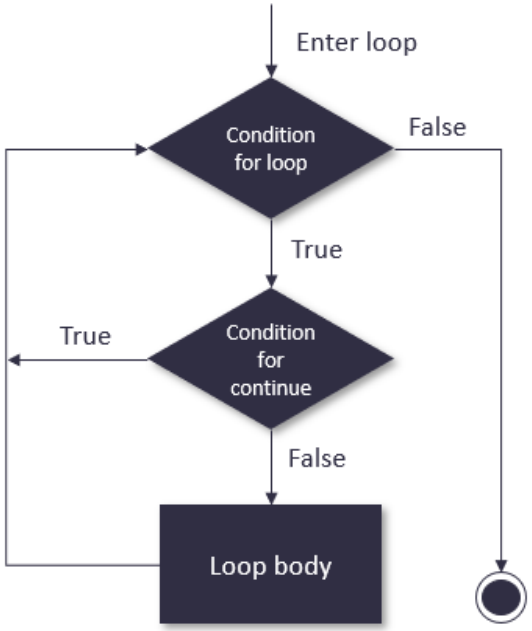
The break Statement

```
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);
    }
}
```

```
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.



`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");
  }
}
```

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

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

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

https://www.runoob.com/java/java-operators.html

The && (Conditional AND) Operator

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

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 (precedence)

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

Evaluate first (associativity)

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*

- 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

- 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

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

expression	!expression
false	true
true	false

The Operators Introduced So Far

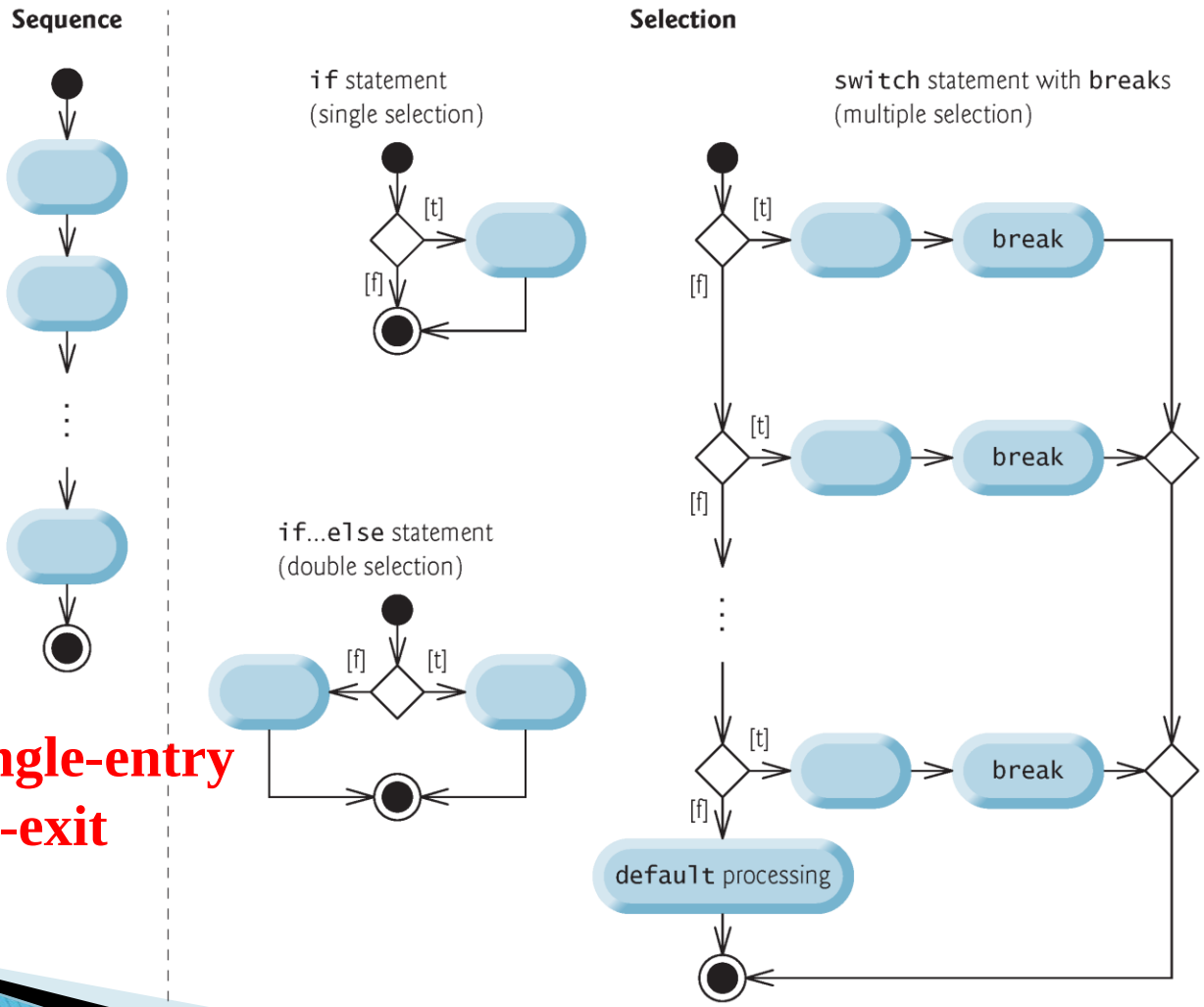
Precedence ↓

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

Objectives

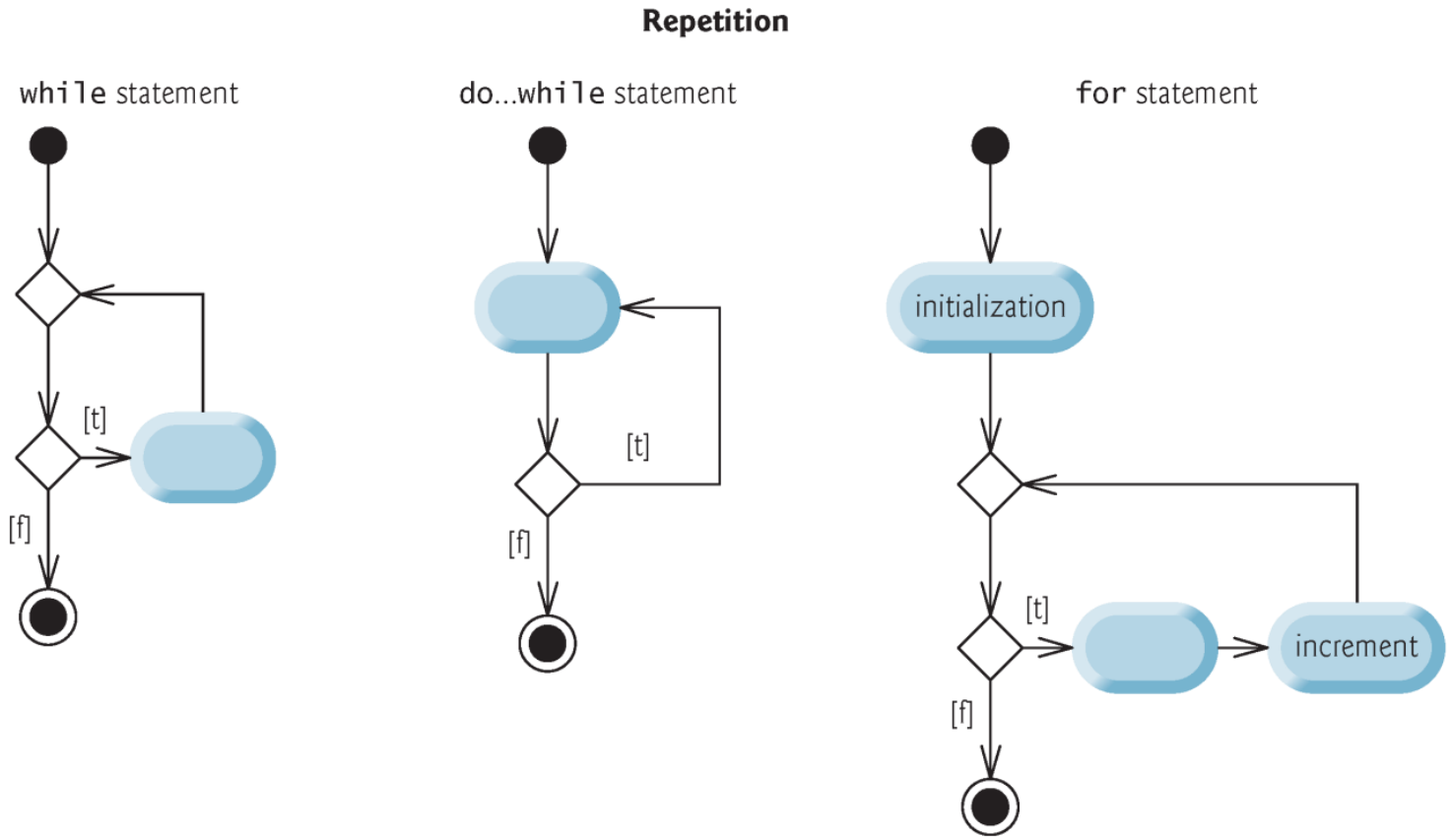
- To use `for` and `do…while` statements
- To use `switch` statement
- To use `continue` and `break` statements
- To use logical operators (逻辑运算符)
- Structured programming

Control Structures Summary



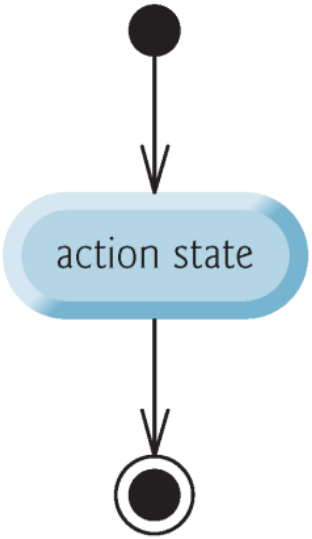
Always single-entry and single-exit

Control Structures Summary



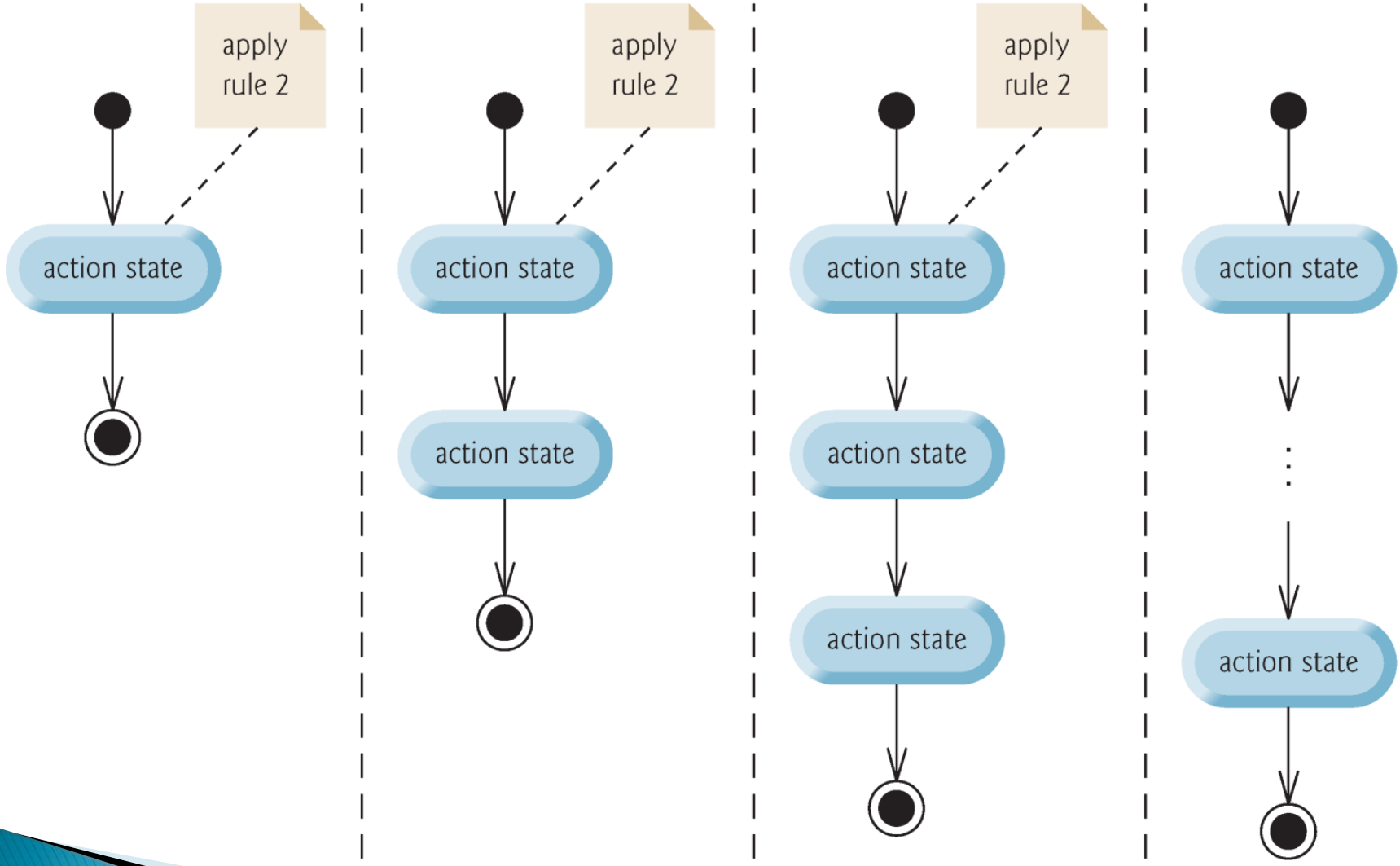
Rules for Forming Structured Programs

- 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.

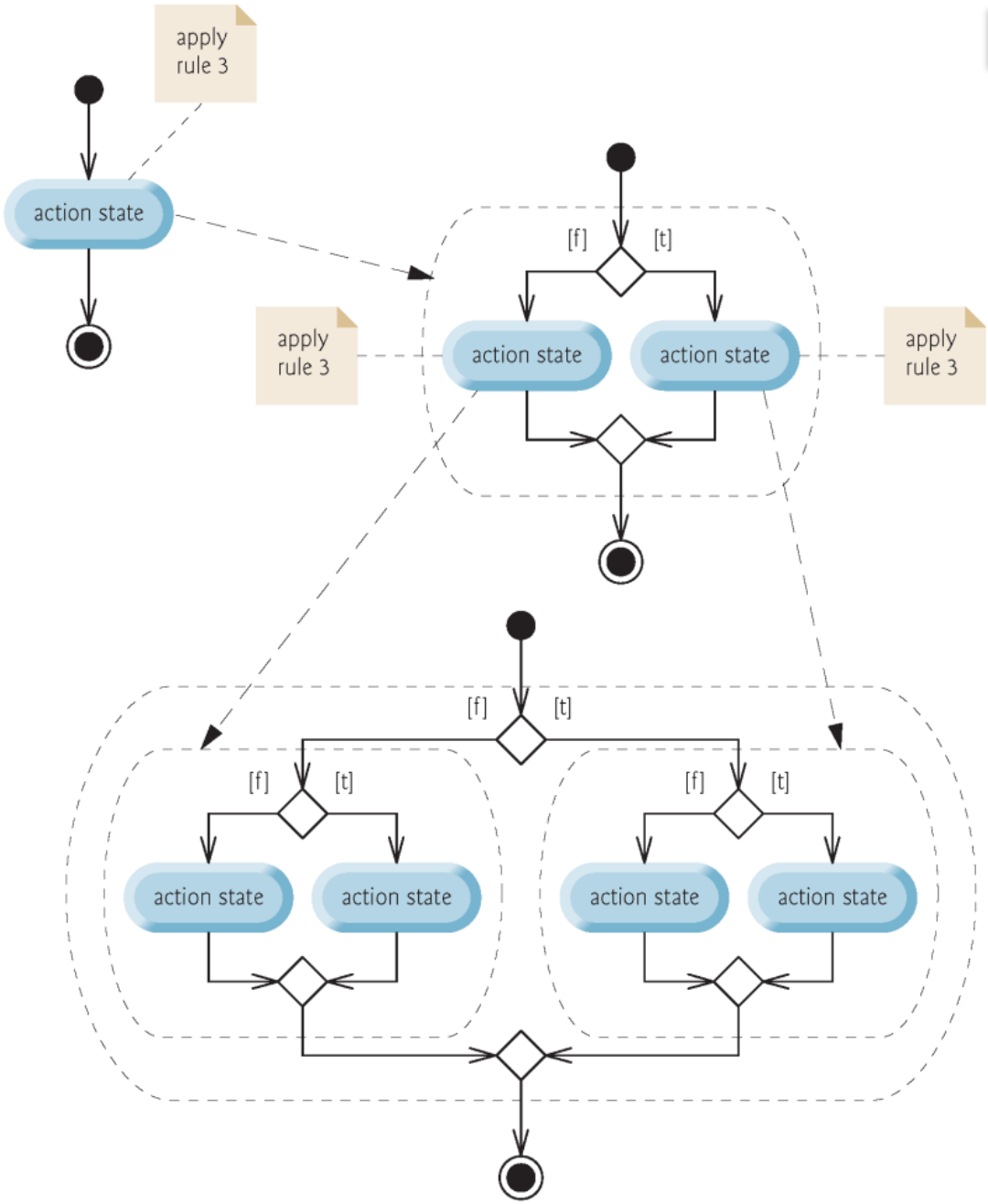


Begin with the simplest activity diagram.

Apply stacking rule

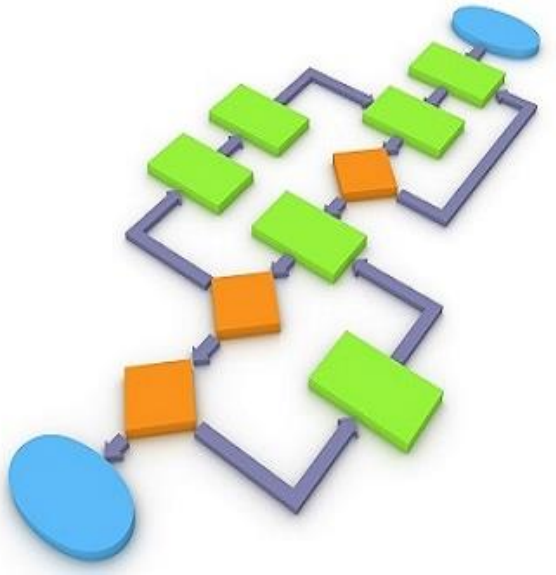


Apply nesting rule



Structured Programming Summary

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



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.

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();
    return;
}
```


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