

# 03 选择结构

## Java程序设计

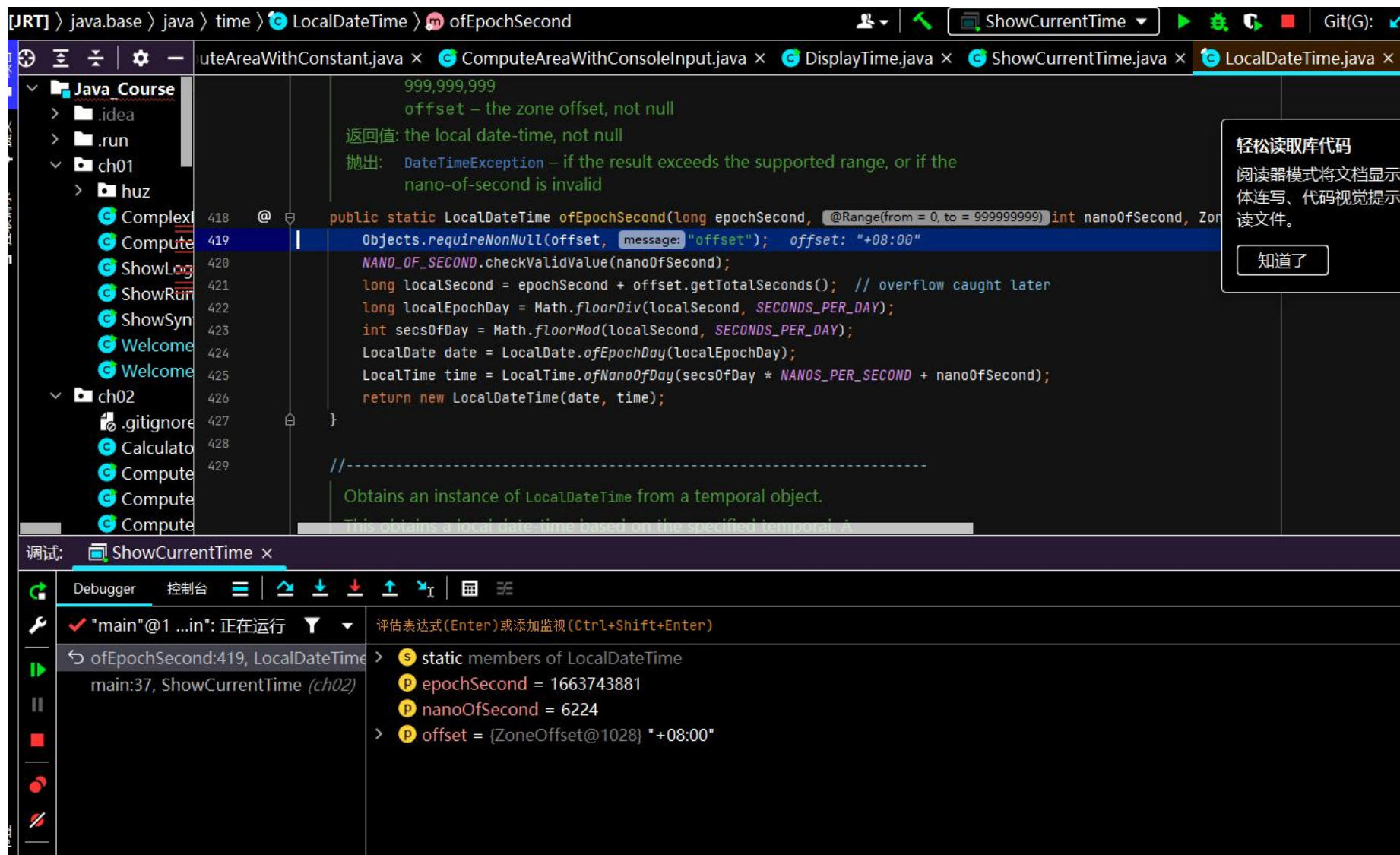
胡铮

[huzheng@bupt.edu.cn](mailto:huzheng@bupt.edu.cn)

2022.08.28



# Ex02 IDE工具的Debug



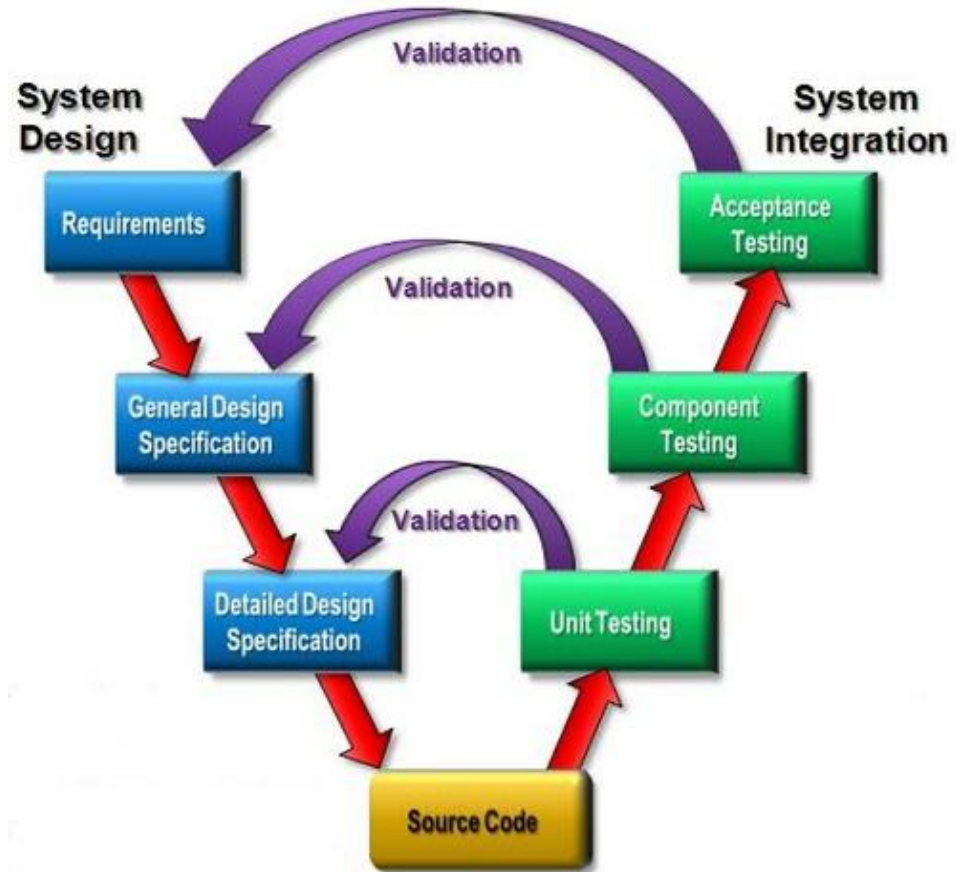


Figure 1: The kindergarten approach to learning

# Debugging

Logic errors are called *bugs*. The process of finding and correcting errors is called debugging. A common approach to debugging is to use a combination of methods to narrow down to the part of the program where the bug is located. You can hand-trace the program (i.e., catch errors by reading the program), or you can insert print statements in order to show the values of the variables or the execution flow of the program. This approach might work for a short, simple program. But for a large, complex program, the most effective approach for debugging is to use a debugger utility.



# Debugger

Debugger is a program that facilitates debugging.  
You can use a debugger to

- ➡ Execute a single statement at a time.
- ➡ Trace into or stepping over a method.
- ➡ Set breakpoints.
- ➡ Display variables.
- ➡ Display call stack.
- ➡ Modify variables.

# Debugging in Eclipse

## Supplement II.G, Learning Java Effectively with Eclipse

	Supplement2fEclipse.pdf	supplement 2f	5 years ago
	Supplement2gEclipseEffecitveTeachi...	add supplement 2g	5 years ago

# The `boolean` Type and Operators

Java provides six comparison operators (also known as relational operators) that can be used to compare two values.

The result of the comparison is a Boolean value: true or false.

```
boolean b = (1 > 2) ;
```

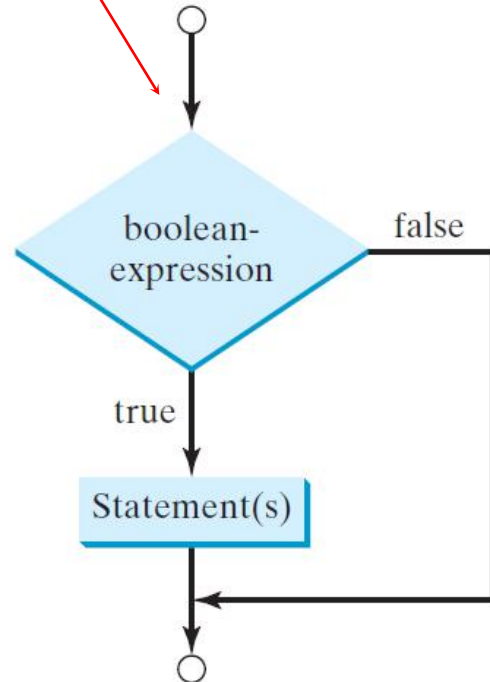
# Relational Operators

Java Operator	Mathematics Symbol	Name	Example (radius is 5)	Result
<	<	less than	<code>radius &lt; 0</code>	<code>false</code>
<=	≤	less than or equal to	<code>radius &lt;= 0</code>	<code>false</code>
>	>	greater than	<code>radius &gt; 0</code>	<code>true</code>
>=	≥	greater than or equal to	<code>radius &gt;= 0</code>	<code>true</code>
==	=	equal to	<code>radius == 0</code>	<code>false</code>
!=	≠	not equal to	<code>radius != 0</code>	<code>true</code>

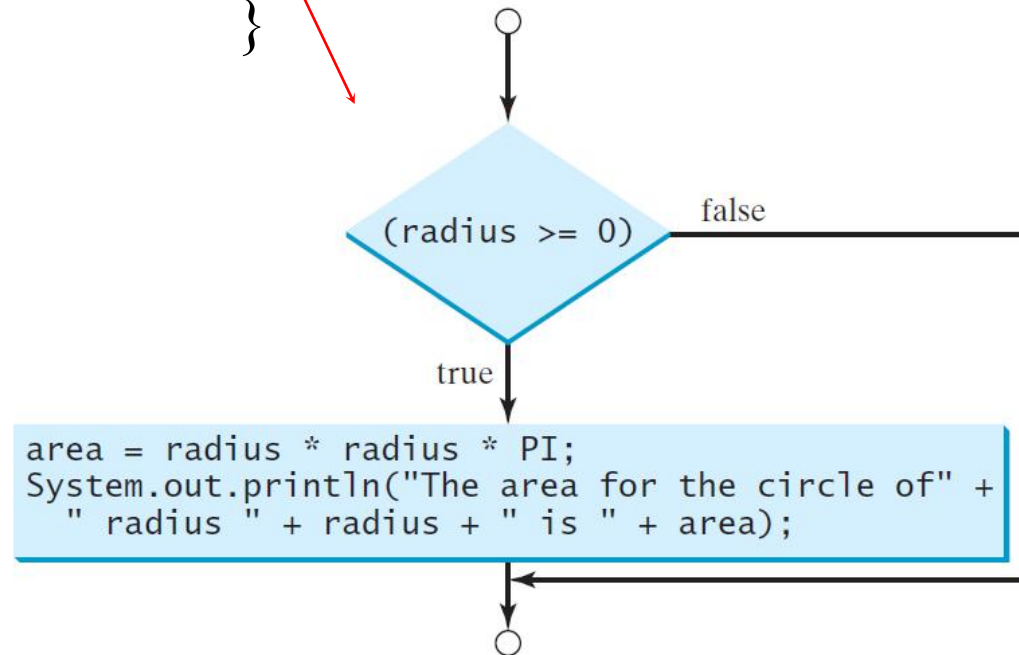


# One-way if Statements

```
if (boolean-expression) {  
    statement(s);  
}
```



```
if (radius >= 0) {  
    area = radius * radius * PI;  
    System.out.println("The area"  
        + " for the circle of radius "  
        + radius + " is " + area);  
}
```



# Note

```
if i > 0 {  
    System.out.println("i is positive");  
}
```

(a) Wrong

```
if (i > 0) {  
    System.out.println("i is positive");  
}
```

(b) Correct

```
if (i > 0) {  
    System.out.println("i is positive");  
}
```

(a)

Equivalent

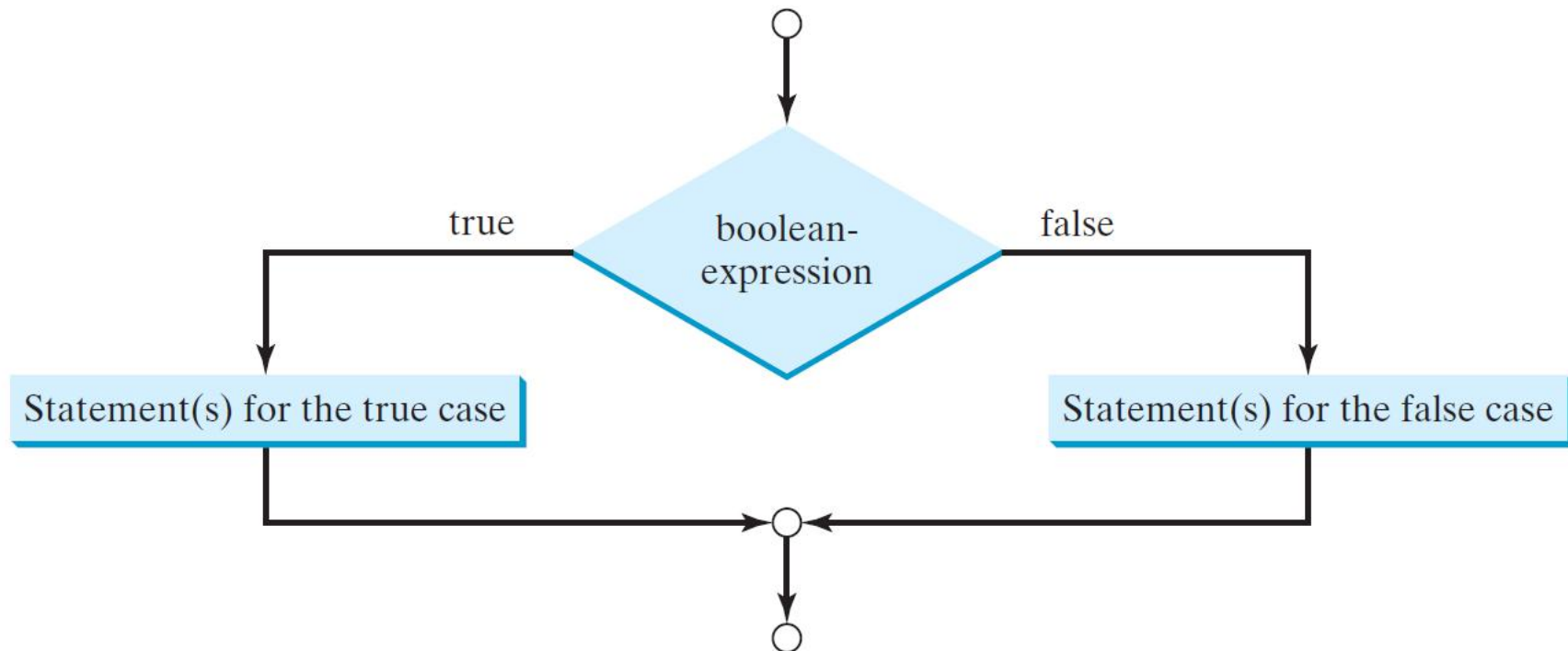
```
if (i > 0)  
    System.out.println("i is positive");
```

(b)

← This is better

# The Two-way `if` Statement

```
if (boolean-expression) {  
    statement(s) -for-the-true-case;  
}  
else {  
    statement(s) -for-the-false-case;  
}
```



# Multiple Alternative if Statements

```
if (score >= 90.0)
    System.out.print("A");
else
    if (score >= 80.0)
        System.out.print("B");
    else
        if (score >= 70.0)
            System.out.print("C");
        else
            if (score >= 60.0)
                System.out.print("D");
            else
                System.out.print("F");
```

(a)

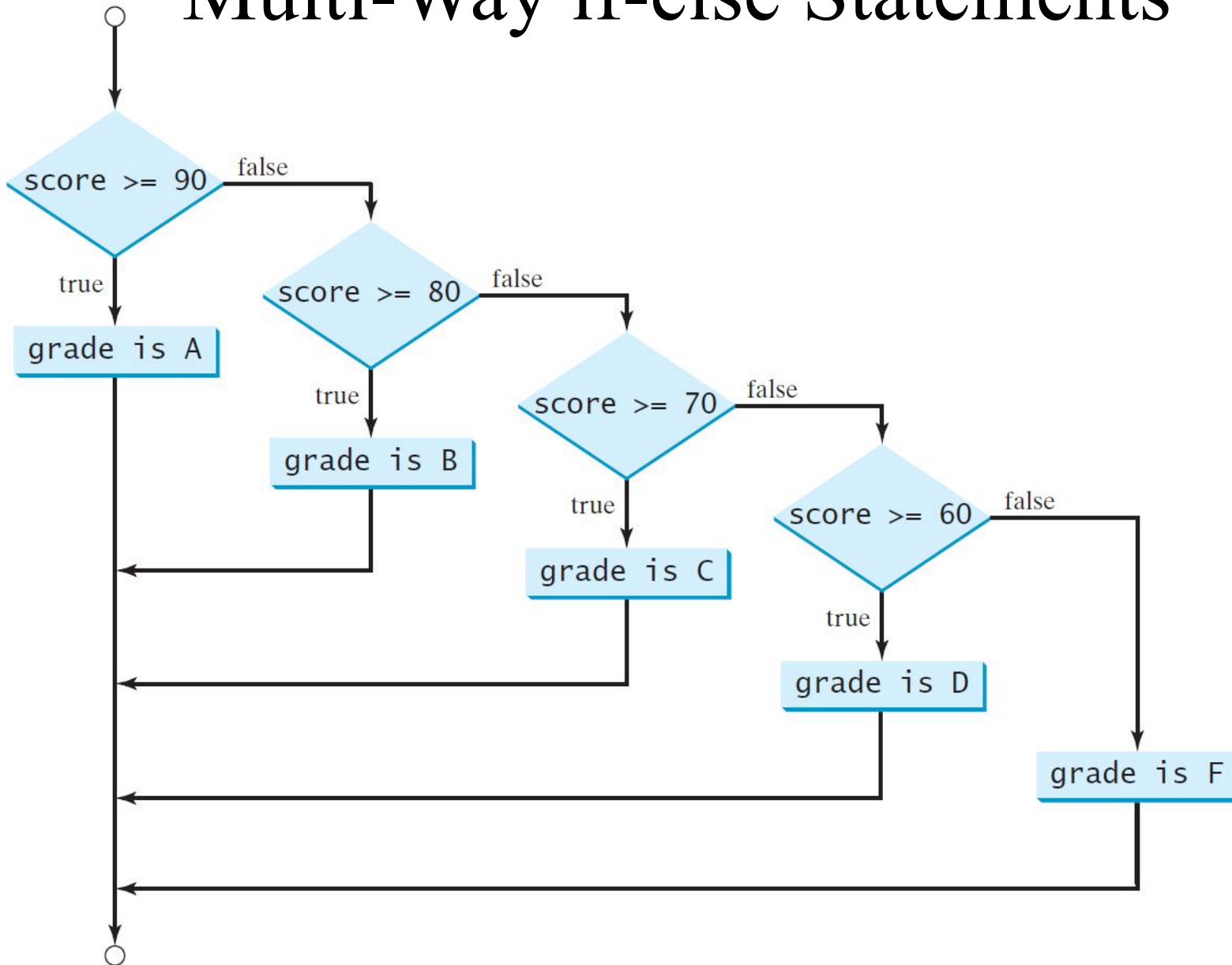
Equivalent

This is better

```
if (score >= 90.0)
    System.out.print("A");
else if (score >= 80.0)
    System.out.print("B");
else if (score >= 70.0)
    System.out.print("C");
else if (score >= 60.0)
    System.out.print("D");
else
    System.out.print("F");
```

(b)

# Multi-Way if-else Statements



# Note

The else clause matches the most recent if clause in the same block.

```
int i = 1, j = 2, k = 3;

if (i > j)
    if (i > k)
        System.out.println("A");
else
    System.out.println("B");
```

(a)

Equivalent

This is better  
with correct  
indentation

```
int i = 1, j = 2, k = 3;

if (i > j)
    if (i > k)
        System.out.println("A");
    else
        System.out.println("B");
```

(b)

# Note, cont.


To force the else clause to match the first if clause, you must add a pair of braces:

```
int i = 1;
int j = 2;
int k = 3;
if (i > j) {
    if (i > k)
        System.out.println("A");
}
else
    System.out.println("B");
```

This statement prints B.

# Common Errors

Adding a semicolon at the end of an if clause is a common mistake.

```
if (radius >= 0);  Wrong
{
    area = radius*radius*PI;
    System.out.println(
        "The area for the circle of radius " +
        radius + " is " + area);
}
```

This mistake is hard to find, because it is not a compilation error or a runtime error, it is a logic error.

This error often occurs when you use the next-line block style.



# TIP

```
if (number % 2 == 0)
    even = true;
else
    even = false;
```

(a)

Equivalent

```
boolean even
    = number % 2 == 0;
```

(b)

# CAUTION

```
if (even == true)
    System.out.println(
        "It is even.");
```

(a)

Equivalent

```
if (even)
    System.out.println(
        "It is even.");
```

(b)

# Problem: An Improved Math Learning Tool

This example creates a program to teach a first grade child how to learn subtractions.

*The program randomly generates two single-digit integers number1 and number2 with number1  $\geq$  number2 and displays a question such as “What is 9 – 2?” to the student. After the student types the answer, the program displays whether the answer is correct.*

ch03/SubtractionQuiz.java

# Problem: Body Mass Index

Body Mass Index (BMI) is a measure of health on weight. It can be calculated by taking your weight in kilograms and dividing by the square of your height in meters. The interpretation of BMI for people 16 years or older is as follows:

BMI	Interpretation
BMI < 18.5	Underweight
18.5 <= BMI < 25.0	Normal
25.0 <= BMI < 30.0	Overweight
30.0 <= BMI	Obese

ch03/ComputeAndInterpretBMI.java

# Problem: Computing Taxes

The US federal personal income tax is calculated based on the filing status and taxable income. There are four filing statuses: single filers, married filing jointly, married filing separately, and head of household. The tax rates for 2009 are shown below.

<i>Marginal Tax Rate</i>	<i>Single</i>	<i>Married Filing Jointly or Qualifying Widow(er)</i>	<i>Married Filing Separately</i>	<i>Head of Household</i>
<b>10%</b>	\$0 – \$8,350	\$0 – \$16,700	\$0 – \$8,350	\$0 – \$11,950
<b>15%</b>	\$8,351 – \$33,950	\$16,701 – \$67,900	\$8,351 – \$33,950	\$11,951 – \$45,500
<b>25%</b>	\$33,951 – \$82,250	\$67,901 – \$137,050	\$33,951 – \$68,525	\$45,501 – \$117,450
<b>28%</b>	\$82,251 – \$171,550	\$137,051 – \$208,850	\$68,526 – \$104,425	\$117,451 – \$190,200
<b>33%</b>	\$171,551 – \$372,950	\$208,851 – \$372,950	\$104,426 – \$186,475	\$190,201 – \$372,950
<b>35%</b>	\$372,951+	\$372,951+	\$186,476+	\$372,951+

# Problem: Computing Taxes, cont.

```
if (status == 0) {  
    // Compute tax for single filers  
}  
else if (status == 1) {  
    // Compute tax for married file jointly  
    // or qualifying widow(er)  
}  
else if (status == 2) {  
    // Compute tax for married file separately  
}  
else if (status == 3) {  
    // Compute tax for head of household  
}  
else {  
    // Display wrong status  
}
```

# Logical Operators

Operator	Name	Description
!	not	logical negation
&&	and	logical conjunction
	or	logical disjunction
^	exclusive or	logical exclusion

# Truth Table for Operator !

p	!p	Example (assume age = 24, weight = 140)
true	false	!(age > 18) is false, because (age > 18) is true.
false	true	!(weight == 150) is true, because (weight == 150) is false.



# Truth Table for Operator &&

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

# Truth Table for Operator $\parallel$

$p_1$	$p_2$	$p_1 \parallel p_2$	Example (assume age = 24, weight = 140)
false	false	false	
false	true	true	$(\text{age} > 34) \parallel (\text{weight} \leq 140)$ is true, because $(\text{age} > 34)$ is false, but $(\text{weight} \leq 140)$ is true.
true	false	true	$(\text{age} > 14) \parallel (\text{weight} \geq 150)$ is true, because $(\text{age} > 14)$ is true.
true	true	true	

# Truth Table for Operator $\wedge$

$p_1$	$p_2$	$p_1 \wedge p_2$	Example (assume age = 24, weight = 140)
false	false	false	$(\text{age} > 34) \wedge (\text{weight} > 140)$ is true, because $(\text{age} > 34)$ is false and $(\text{weight} > 140)$ is false.
false	true	true	$(\text{age} > 34) \wedge (\text{weight} \geq 140)$ is true, because $(\text{age} > 34)$ is false but $(\text{weight} \geq 140)$ is true.
true	false	true	$(\text{age} > 14) \wedge (\text{weight} > 140)$ is true, because $(\text{age} > 14)$ is true and $(\text{weight} > 140)$ is false.
true	true	false	

Here is a program that checks

TestBooleanOperators.java

- whether a number is divisible by 2 and 3,
- whether a number is divisible by 2 or 3,
- and whether a number is divisible by 2 or 3 but not both:

```
System.out.println("Is " + number + " divisible by 2 and 3? " +
```

```
((number % 2 == 0) [填空1] (number % 3 == 0)));
```

```
System.out.println("Is " + number + " divisible by 2 or 3? " +
```

```
((number % 2 == 0) [填空2] (number % 3 == 0)));
```

```
System.out.println("Is " + number +
```

```
" divisible by 2 or 3, but not both? " +
```

```
((number % 2 == 0) [填空3] (number % 3 == 0)));
```

作答

# The & and | Operators

Both & and && are the boolean and operators.

Both | and || are the boolean or operators.

The difference between the two is that && and || are the short-circuit operator.

```
if( false & true ) // in this, both will be evaluated.
```

```
if( false && true ) // only the false will be evaluated.
```

```
int i = 0;
```

```
if( false & (i++) == 2) // i will now = 1 because both were evaluated.
```

```
int i = 0;
```

```
if( false && (i++) == 2) //i will still be 0 because the second  
    expression was not evaluated.
```

# Problem: Determining Leap Year?

This program first prompts the user to enter a year as an int value and checks if it is a leap year.

A year is a leap year if it **is divisible by 4** but **not by 100**, or it is divisible by 400.

```
(year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)
```

LeapYear.java

# Problem: Lottery

Write a program that randomly generates a lottery of a two-digit number, prompts the user to enter a two-digit number, and determines whether the user wins according to the following rule:

- If the user input matches the lottery in exact order, the award is \$10,000.
- If the user input matches the lottery, the award is \$3,000.
- If one digit in the user input matches a digit in the lottery, the award is \$1,000.

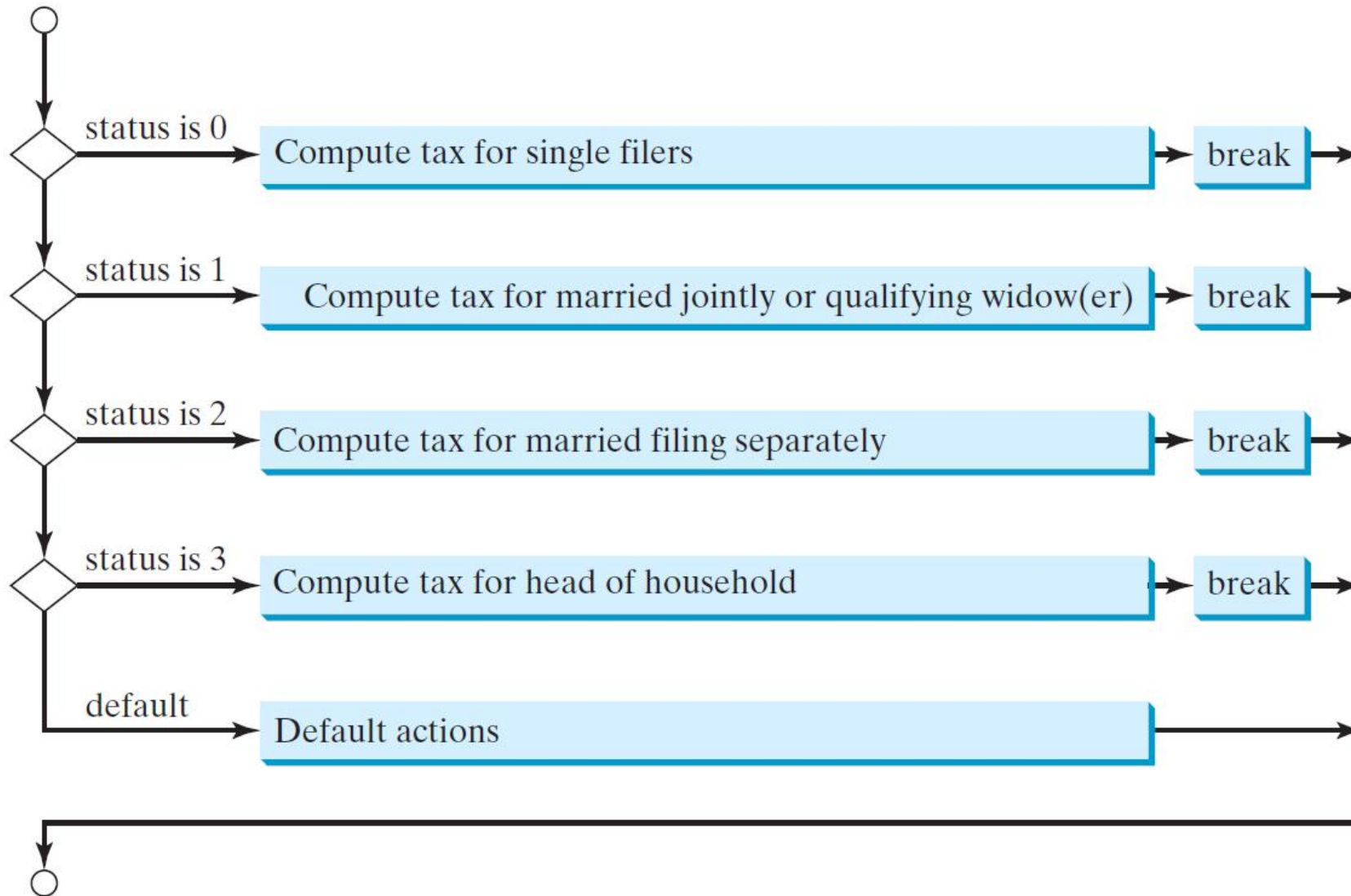
ch03/Lottery.java

# switch Statements

```
switch (status) {                                     ch03/ChineseZodiac.java
    case 0: compute taxes for single filers;
        break;
    case 1: compute taxes for married file jointly;
        break;
    case 2: compute taxes for married file separately;
        break;
    case 3: compute taxes for head of household;
        break;
    default: System.out.println("Errors: invalid status");
        System.exit(1);
}
```



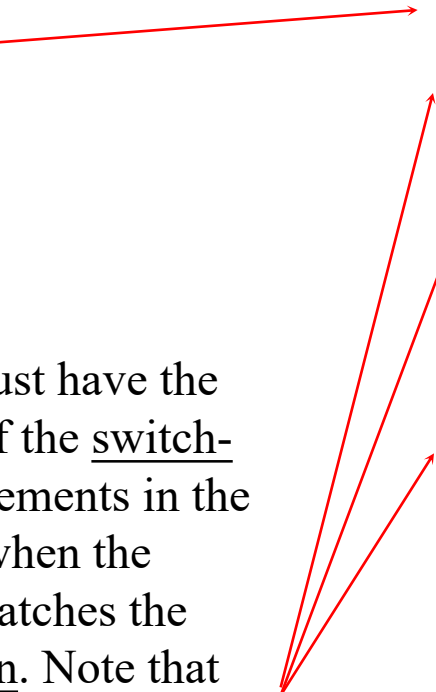
# switch Statement Flow Chart



# switch Statement Rules

The switch-expression must yield a value of char, byte, short, or int primitive types, or String types, or enumerated types (Enum Types) must always be enclosed in parentheses.

The value1, ..., and valueN must have the same data type as the value of the switch-expression. The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression. Note that value1, ..., and valueN are constant expressions, meaning that they cannot contain variables in the expression, such as  $1 + x$ .



```
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;  
}
```

ch03/TestSwitchExpression.java

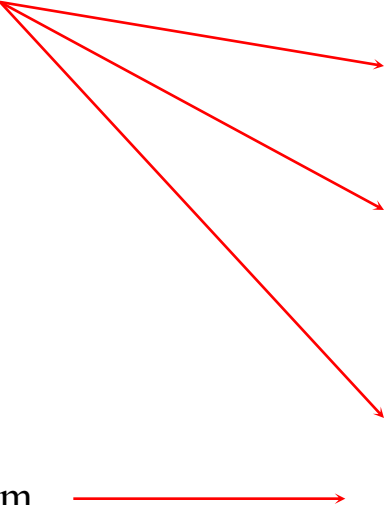
ch03/EnumTest.java

# switch Statement Rules

The keyword break is optional, but it should be used at the end of each case in order to terminate the remainder of the switch statement. If the break statement is not present, the next case statement will be executed.

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

```
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;  
}
```



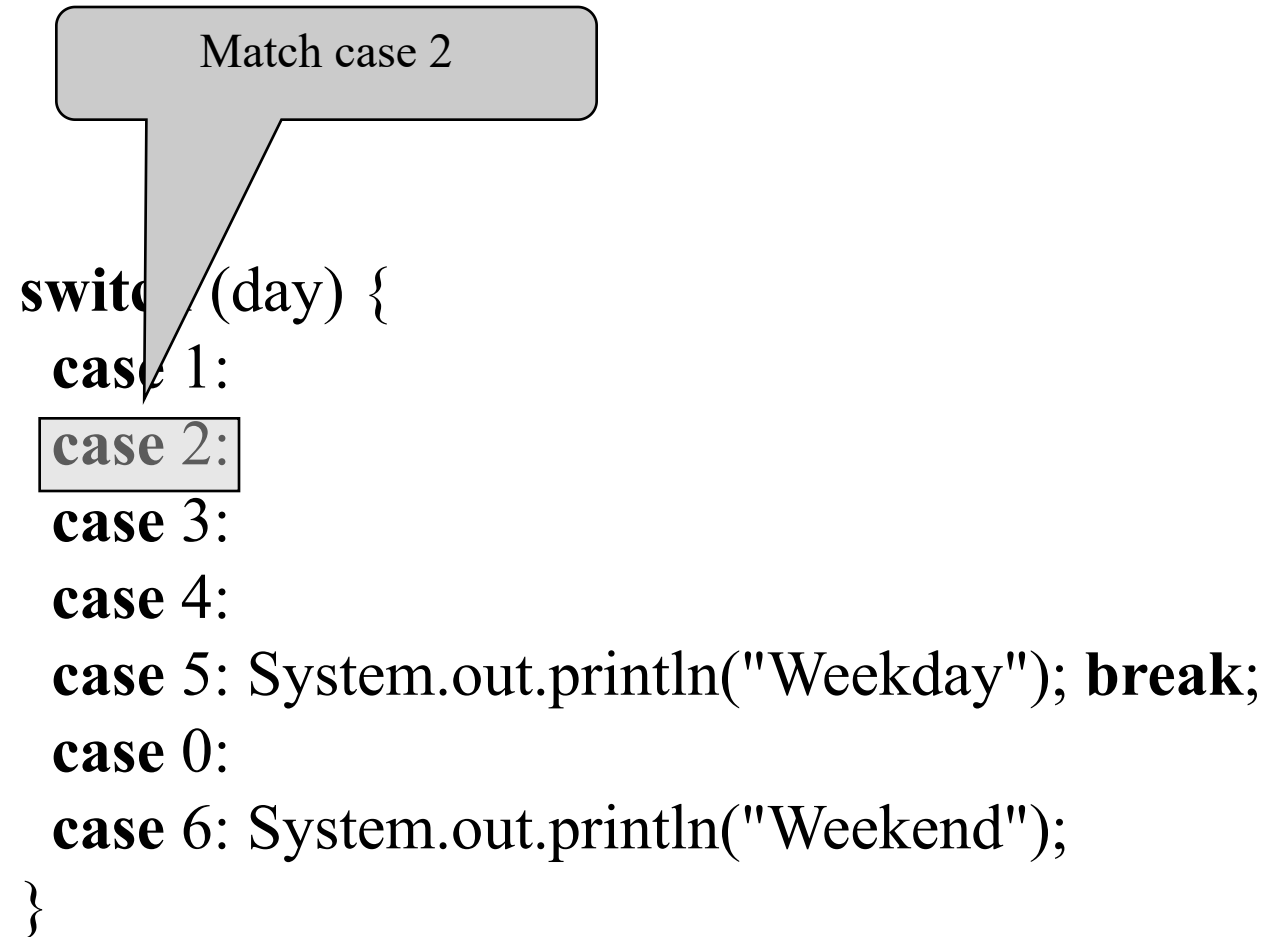
When the value in a **case** statement matches the value of the **switch-expression**, the statements *starting from this case* are executed until either a **break** statement or the end of the **switch** statement is reached.

# Trace switch statement

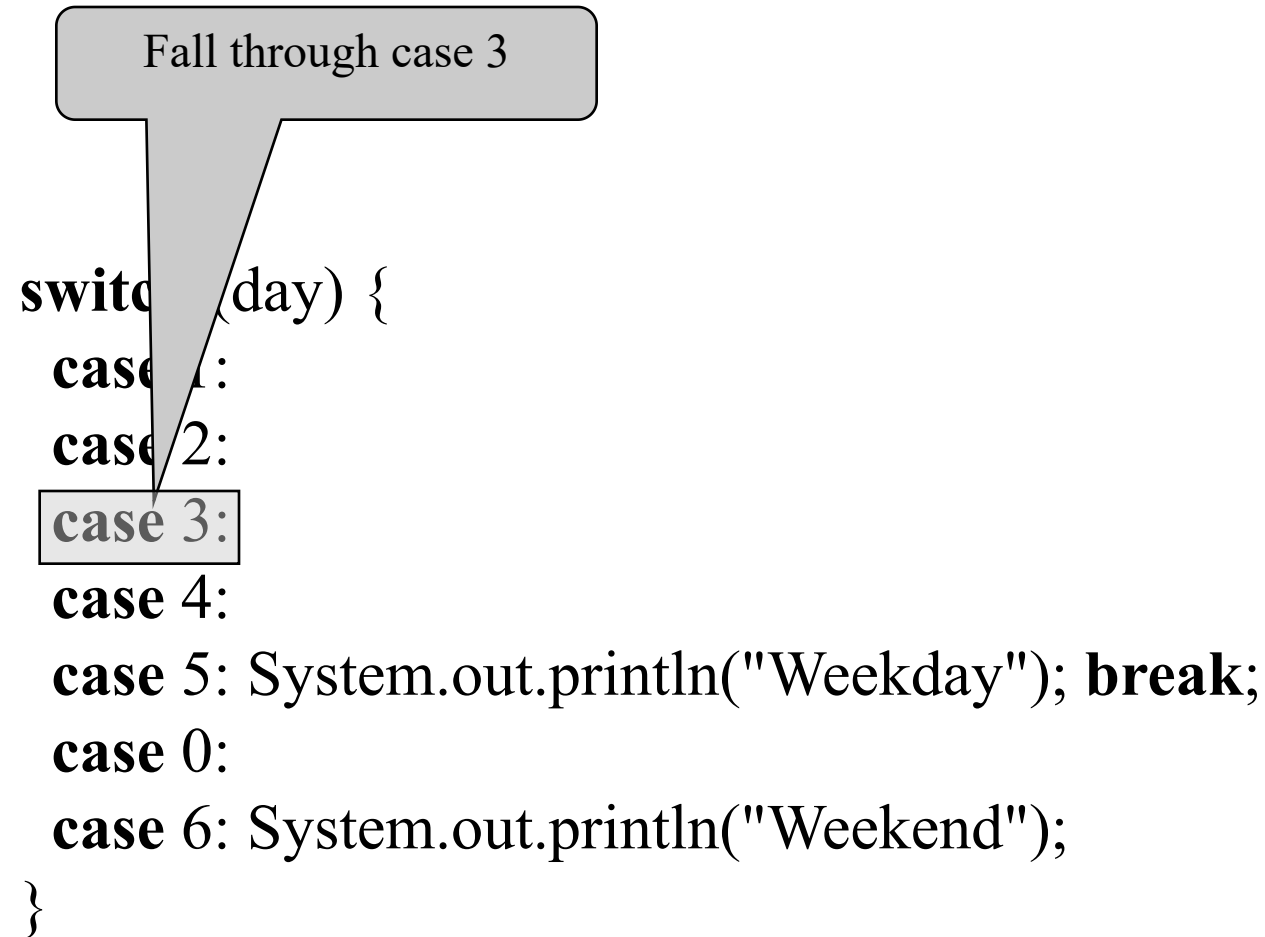
Suppose day is 2:

```
switch (day) {  
  case 1:  
  case 2:  
  case 3:  
  case 4:  
  case 5: System.out.println("Weekday"); break;  
  case 0:  
  case 6: System.out.println("Weekend");  
}
```

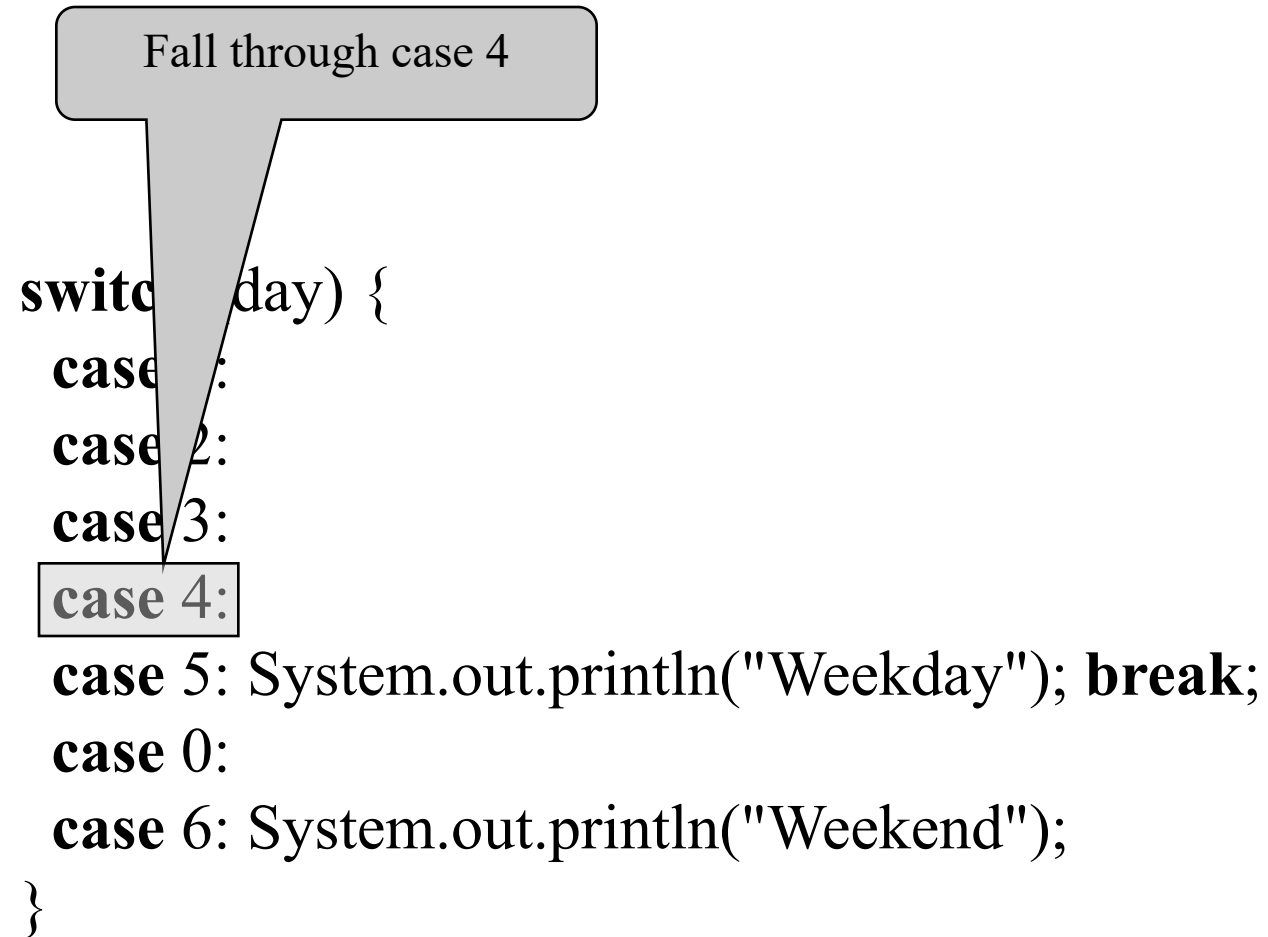
# Trace switch statement



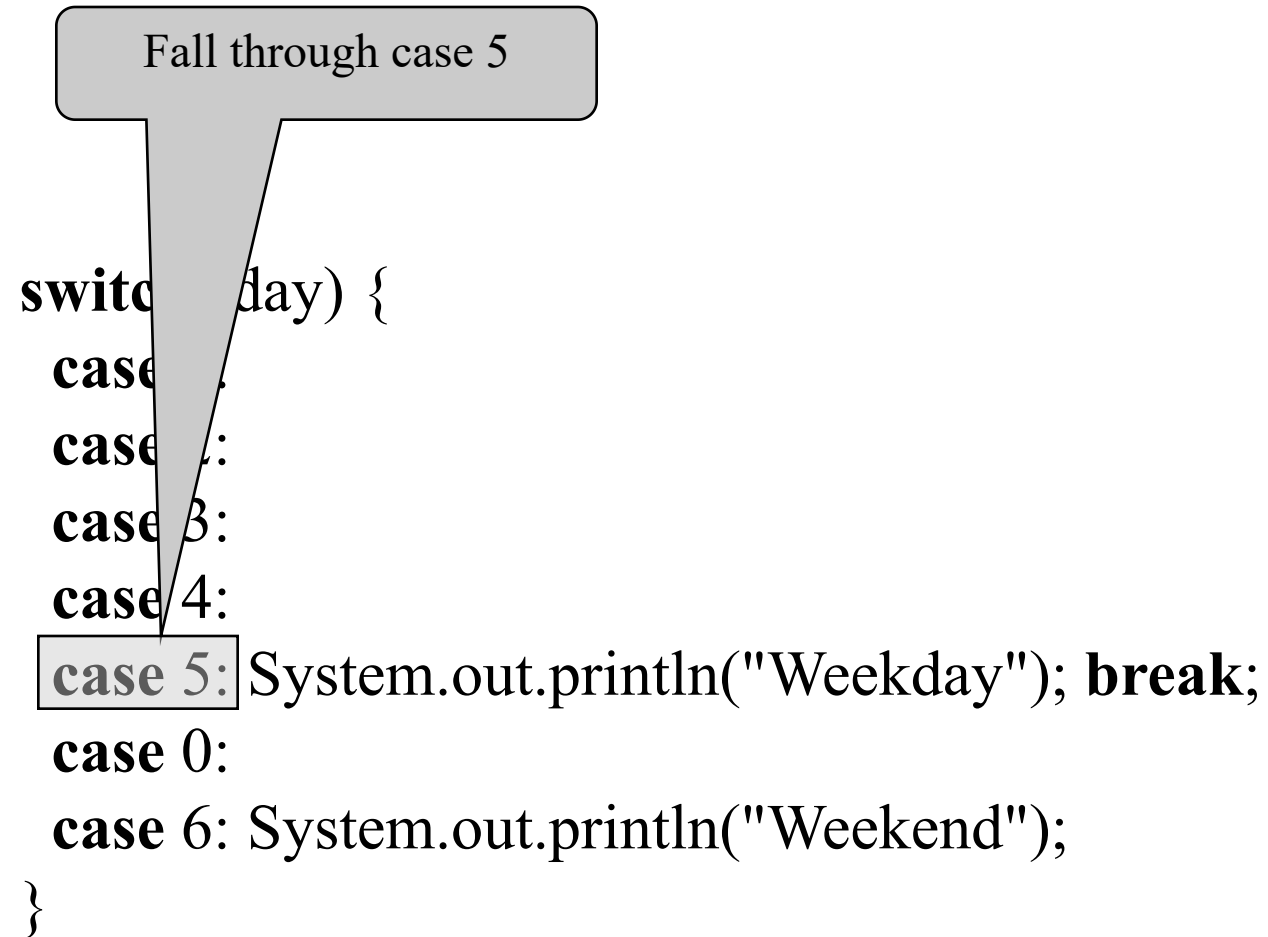
# Trace switch statement



# Trace switch statement



# Trace switch statement





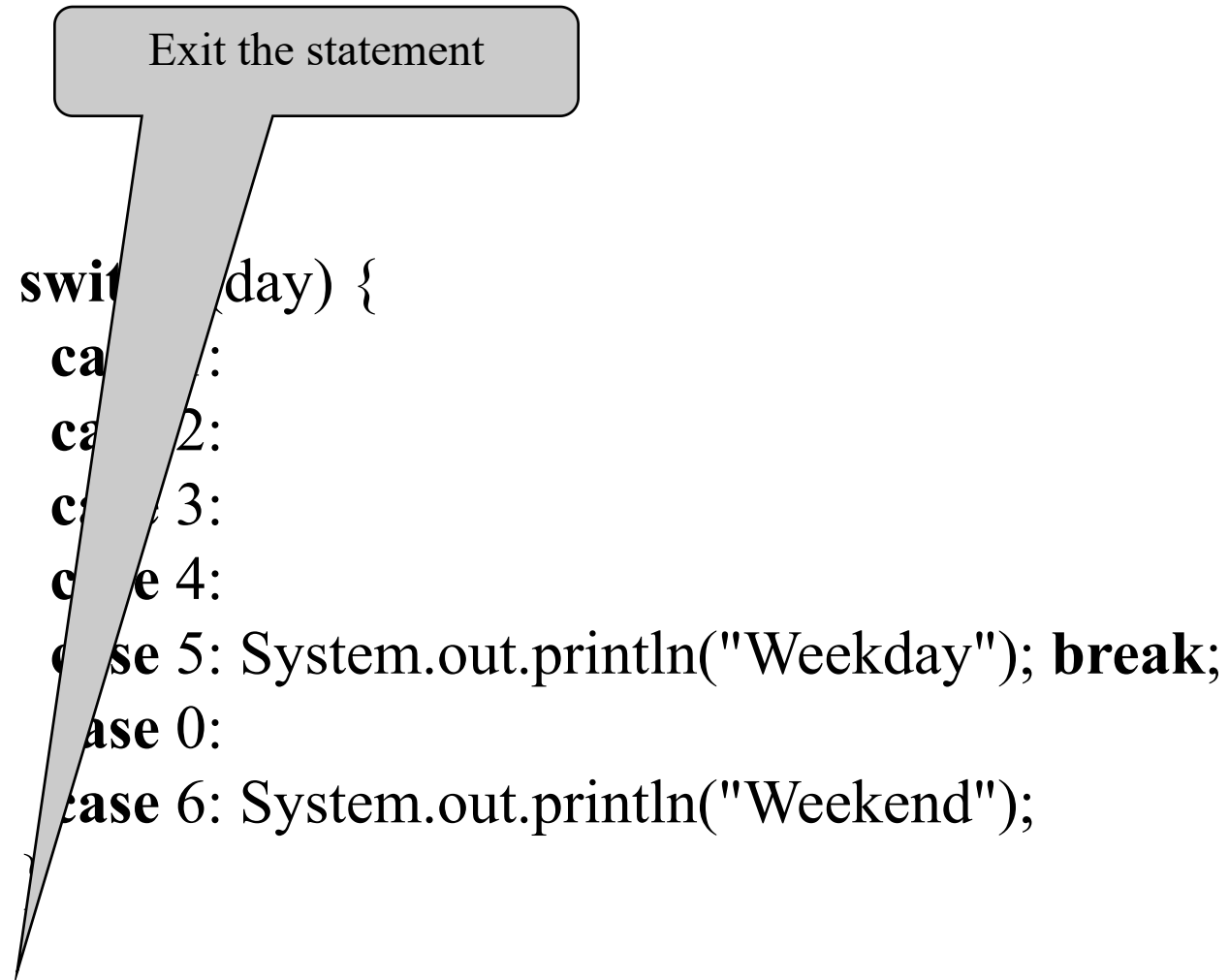
# Trace switch statement

Encounter break



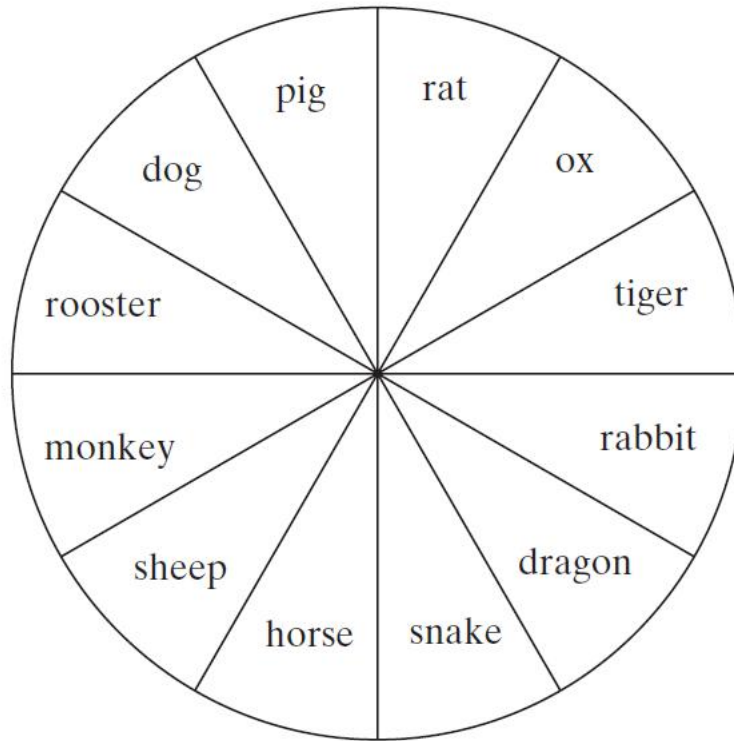
```
switch (day) {  
  case 1:  
  case 2:  
  case 3:  
  case 4:  
  case 5: System.out.println("Weekday"); break;  
  case 0:  
  case 6: System.out.println("Weekend");  
}
```

# Trace switch statement



# Problem: Chinese Zodiac

Write a program that prompts the user to enter a year and displays the animal for the year.



$\text{year} \% 12 =$  {  
0: monkey  
1: rooster  
2: dog  
3: pig  
4: rat  
5: ox  
6: tiger  
7: rabbit  
8: dragon  
9: snake  
10: horse  
11: sheep

ch03/ChineseZodiac.java

# Conditional Expressions

```
if (x > 0)
```

```
    y = 1
```

```
else
```

```
    y = -1;
```

is equivalent to

```
y = (x > 0) ? 1 : -1;
```

```
(boolean-expression) ? expression1 : expression2
```

Ternary operator

Binary operator

Unary operator

# Conditional Operator

```
if (num % 2 == 0)
    System.out.println(num + "is even");
else
    System.out.println(num + "is odd");
```

```
System.out.println(
    (num % 2 == 0)? num + "is even" :
    num + "is odd");
```

# Operator Precedence

- ☞ `var++`, `var--`
- ☞ `+`, `-` (Unary plus and minus), `++var`, `--var`
- ☞ `(type)` Casting
- ☞ `!` (Not)
- ☞ `*`, `/`, `%` (Multiplication, division, and remainder)
- ☞ `+`, `-` (Binary addition and subtraction)
- ☞ `<`, `<=`, `>`, `>=` (Relational operators)
- ☞ `==`, `!=`; (Equality)
- ☞ `^` (Exclusive OR)
- ☞ `&&` (Conditional AND) Short-circuit AND
- ☞ `||` (Conditional OR) Short-circuit OR
- ☞ `=`, `+=`, `-=`, `*=`, `/=`, `%=` (Assignment operator)

# Operator Precedence and Associativity

The expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule.

If operators with the same precedence are next to each other, their associativity determines the order of evaluation. All binary operators except assignment operators are left-associative.

# Operator Associativity

When two operators with the same precedence are evaluated, the *associativity* of the operators determines the order of evaluation. All binary operators except assignment operators are *left-associative*.

$a - b + c - d$  is equivalent to  $((a - b) + c) - d$

Assignment operators are *right-associative*. Therefore, the expression

$a = b += c = 5$  is equivalent to  $a = (b += (c = 5))$



# Example

Applying the operator precedence and associativity rule, the expression  $3 + 4 * 4 > 5 * (4 + 3) - 1$  is evaluated as follows:

