

Chapter 3

Selections

Programming I --- Ch. 3

1

Objectives

- To declare **boolean** variables and write Boolean expressions using relational operators (<, <=, >, >=, ==, !=)
- To implement selection control using one-way **if** statements, two-way **if-else** statements, nested **if** and multi-way **if** statements
- To generate random numbers using the **Math.random()** method
- To combine conditions using logical operators (!, &&, ||, and ^)
- To implement selection control using **switch** statements
- To examine the rules governing operator precedence and associativity
- To apply common techniques to debug errors

Programming I --- Ch. 3

2

Motivations

- If you assigned a negative value for radius in Listing 2.2, `ComputeAreaWithConsoleInput.java`, the program would print an invalid result.
- If the radius is negative, you don't want the program to compute the area.
- How can you deal with this situation?

boolean Data Type

- The **boolean** data type declares a variable with the value either **true** or **false**.
- Here, **true** and **false** are literals, just like a number such as **10**. They are treated as reserved words and cannot be used as identifiers in the program.
- A variable that holds a Boolean value is known as a *Boolean variable*.
- For example, `boolean b = (1 > 2);`

Relational Operators

- Java provides six *relational operators* (also known as *comparison operators*), shown in Table 3.1, which can be used to compare two values (assume radius is 5 in the table).

TABLE 3.1 Relational Operators

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

Note that The equality testing operator is two equal signs (==), not a single equal sign (=). The latter symbol is for assignment.

An example on boolean data type

- Suppose you want to develop a program to randomly generate two single-digit integers, **number1** and **number2**, and displays a question such as "What is 1 + 7?,". After the student types the answer, the program displays a message to indicate whether it is true or false.

LISTING 3.1 AdditionQuiz.java

```

1 import java.util.Scanner;
2
3 public class AdditionQuiz {
4     public static void main(String[] args) {
5         int number1 = (int)(System.currentTimeMillis() % 10);
6         int number2 = (int)(System.currentTimeMillis() / 7 % 10);
7
8         // Create a Scanner
9         Scanner input = new Scanner(System.in);
10
11         System.out.print(
12             "What is " + number1 + " + " + number2 + "? ";
13
14         int answer = input.nextInt();
15
16         System.out.println(
17             number1 + " + " + number2 + " = " + answer + " is " +
18             (number1 + number2 == answer));
19     }
20 }
```

This method returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC (coordinated universal time).

generate number1
generate number2

show question

display result

Selection Statements

- Java provides *selection statements*: statements that let you choose actions with alternative courses.
- Selection statements use conditions that are Boolean expressions. A *Boolean expression* is an expression that evaluates to a *Boolean value*: **true** or **false**.
- If you enter a negative value for **radius** in Listing 2.2, `ComputeAreaWithConsoleInput.java`, the program displays an invalid result. If the radius is negative, you don't want the program to compute the area. You can use the following selection statement to replace lines 12–17 in Listing 2.2:

```
if (radius < 0) {
    System.out.println("Incorrect input");
}
else {
    area = radius * radius * 3.14159;
    System.out.println("Area is " + area);
}
```

Programming I --- Ch. 3

7

if Statements

- An *if statement* is a construct that enables a program to specify alternative paths of execution.
- Java has several types of selection statements: one-way **if** statements, two-way **if-else** statements, nested **if** statements, multi-way **if-else** statements, **switch** statements, and conditional expressions.
- A one-way **if** statement executes an action if and only if the condition is **true**. The syntax for a one-way **if** statement is:

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

Programming I --- Ch. 3

8

if Statements (cont'd)

- The **boolean-expression** is enclosed in parentheses. For example, the code in (a) is wrong. It should be corrected, as shown in (b).

<pre>if i > 0 { System.out.println("i is positive"); }</pre>	<pre>if (i > 0) { System.out.println("i is positive"); }</pre>
(a) Wrong	(b) Correct

- The block braces can be omitted if they enclose a single statement only. However, omitting braces is prone to errors. It is a common mistake to forget the braces when you go back to modify the code that omits the braces.

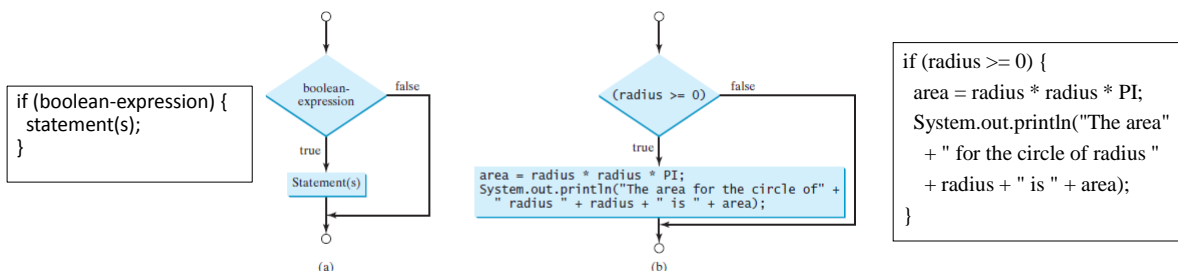
<pre>if (i > 0) { System.out.println("i is positive"); }</pre>	Equivalent	<pre>if (i > 0) System.out.println("i is positive");</pre>
(a)		(b)

Programming I --- Ch. 3

9

Flowchart of how Java executes the syntax of an if statement

- A **flowchart** is a diagram that describes an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows.
- Process operations are represented in these boxes, and arrows connecting them represent the flow of control.
- A diamond box denotes a Boolean condition and a rectangle box represents statements.

FIGURE 3.1 An if statement executes statements if the **boolean-expression** evaluates to **true**.

programming I --- Ch. 3

10

Two-way if-else Statements

- An **if-else** statement decides the execution path based on whether the condition is true or false. Below is the syntax for a two-way **if-else** statement:

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

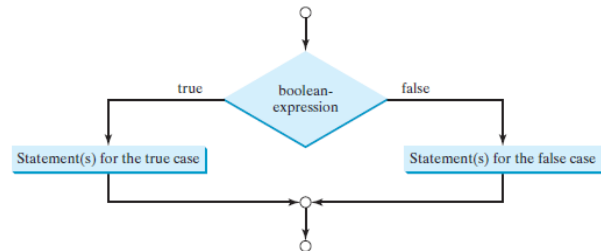


FIGURE 3.2 An **if-else** statement executes statements for the true case if the **Boolean-expression** evaluates to **true**; otherwise, statements for the false case are executed.

Two-way if-else Statements

- An example of using the **if-else** statement. The example checks whether a number is even or odd, as follows:

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

Nested if and Multi-way if-else Statements

- An **if** statement can be inside another **if** statement to form a **nested if** statement.
- There is no limit to the depth of the nesting. The nested **if** statement can be used to implement multiple alternatives.
- The statement given in Figure 3.3a, for instance, prints a letter grade according to the score, with multiple alternatives.

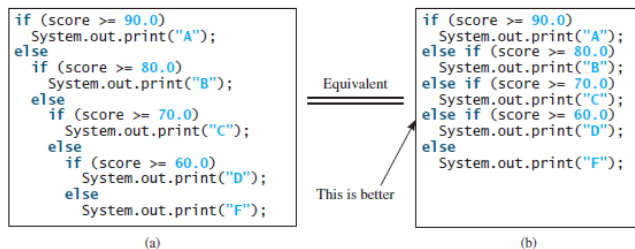
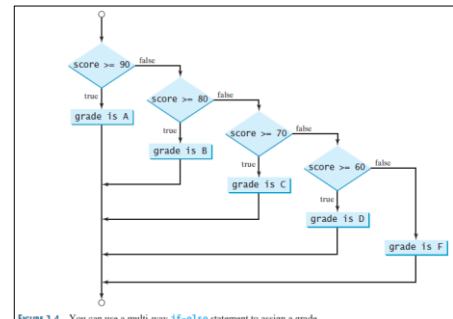


FIGURE 3.3 A preferred format for multiple alternatives is shown in (b) using a multi-way **if-else** statement.

Programming I --- Ch. 3



13

animation

Trace if-else statement

Suppose score is 70.0

The condition is false

```

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");
  
```

animation

Trace if-else statement

Suppose score is 70.0

The condition is false

```

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");

```

15

animation

Trace if-else statement

Suppose score is 70.0

The condition is true

```

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");

```

16

animation

Trace if-else statement

Suppose score is 70.0

grade is C

```
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");
```

17

animation

Trace if-else statement

Suppose score is 70.0

Exit the if statement

```
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");
```

18

Dangling else Ambiguity

- The code in (a) below has two `if` clauses and one `else` clause. Which `if` clause is matched by the `else` clause?
- The indentation indicates that the `else` clause matches the first `if` clause. However, the `else` clause actually matches the second `if` clause.

```
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)

What is the output of
this fragment of code?

- The else clause matches the most recent if clause in the same block.
- What is the output of this fragment of code now that a pair of `{ }` is added?

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

Programming I --- Ch. 3

19

Dangling else Ambiguity (con't)

Nothing is printed from the preceding statement. 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");
```

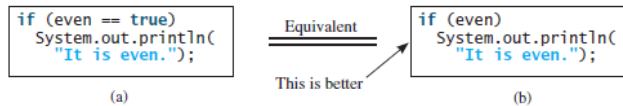
This statement prints B.

Programming I --- Ch. 3

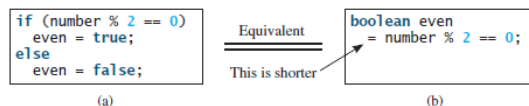
20

Common Pitfalls using if statement

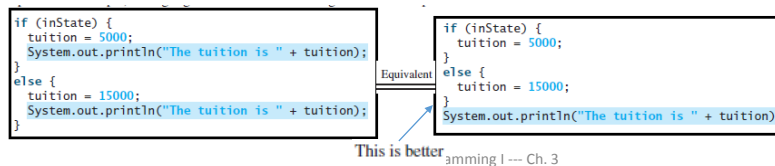
• Redundant Testing of Boolean Values



• Simplifying Boolean Variable Assignment



• Avoiding Duplicate Code in Different Cases



Programming I --- Ch. 3

21

Using `hasNextInt()` to prevent invalid input which causes runtime error

- `hasNextInt()` returns a boolean to indicate if the next input can be safely read as an integer.

LISTING 3.1 AdditionQuiz.java

```
1 import java.util.Scanner;
2
3 public class AdditionQuiz {
4     public static void main(String[] args) {
5         int number1 = (int)(System.currentTimeMillis() % 10);
6         int number2 = (int)(System.currentTimeMillis() / 7 % 10);
7
8         // Create a Scanner
9         Scanner input = new Scanner(System.in);
10
11         System.out.print(
12             "What is " + number1 + " + " + number2 + "? ");
13
14         int answer = input.nextInt();
15
16         System.out.println(
17             number1 + " + " + number2 + " = " + answer + " is " +
18             (number1 + number2 == answer));
19     }
20 }
```

If user just presses "Enter" without any value, the program keeps waiting for input.
Or if user enters 'a' as input, runtime error occurs: `InputMismatchException`.

```
int answer;
if (input.hasNextInt()) {
    answer = input.nextInt();
    System.out.println("Answer is " + (number1 + number2 ==
    answer));
}
else
    System.out.println("Answer is of type integer.");
```

Programming I --- Ch. 3

22

Testing for valid input

- A Scanner has methods to see what the next token will be.

Method	Description
<code>hasNextInt()</code>	returns <code>true</code> if there is a next token and it can be read as an <code>int</code>
<code>hasNextDouble()</code>	returns <code>true</code> if there is a next token and it can be read as a <code>double</code>

- These methods do not actually consume input. It just gives information about what the input is waiting.

Generating Random Numbers

- You can use **`Math.random()`** to obtain a random double value **`d`** such that $0.0 \leq d < 1.0$
- Thus, **`(int)(Math.random() * 10)`** returns a random single-digit integer (i.e., a number between **`0`** and **`9`**).
- To generate a random integer within a range:


```
// define the range
int max = 10;
int min = 1;
int range = max - min + 1; //upperbound is inclusive; if without +1, upperbound is exclusive

// generate random integers within the range
int rand = (int)(Math.random() * range) + min;
```
- Read **LISTING 3.3** SubtractionQuiz.java for an example on generating random numbers with `Math.random()`, as shown in the next slide.

Generating Random Numbers: an example

LISTING 3.3 SubtractionQuiz.java

```

1  import java.util.Scanner;
2
3  public class SubtractionQuiz {
4      public static void main(String[] args) {
5          // 1. Generate two random single-digit integers
6          int number1 = (int)(Math.random() * 10);
7          int number2 = (int)(Math.random() * 10);
8
9          // 2. If number1 < number2, swap number1 with number2
10         if (number1 < number2) {
11             int temp = number1;
12             number1 = number2;
13             number2 = temp;
14         }
15
16         // 3. Prompt the student to answer "What is number1 - number2?"
17         System.out.print
18             ("What is " + number1 + " - " + number2 + "? ");
19         Scanner input = new Scanner(System.in);
20         int answer = input.nextInt();
21
22         // 4. Grade the answer and display the result
23         if (number1 - number2 == answer)
24             System.out.println("You are correct!");
25         else {
26             System.out.println("Your answer is wrong.");
27             System.out.println(number1 + " - " + number2 +
28                 " should be " + (number1 - number2));
29         }
30     }
31 }

```

25

Logical Operators

- The *logical operators*, also known as *Boolean operators*, **!**, **&&**, **||**, and **^** operate on Boolean values to create a new Boolean value.
- Table 3.3 lists the Boolean operators.
 - **!**: negates **true** to **false** and **false** to **true**
 - **&&**: The and (**&&**) of two Boolean operands is **true** if and only if both operands are **true**.
 - **||**: The or (**||**) of two Boolean operands is **true** if at least one of the operands is **true**.
 - The exclusive or (**^**) of two Boolean operands is **true** if and only if the two operands have different Boolean values. Note that **p1 ^ p2** is the same as **p1 != p2**.

TABLE 3.3 Boolean Operators

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

Truth Table for Operator &&

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 and (&&) of two Boolean operands is **true** if and only if both operands are **true**.

Truth Table for Operator ||

TABLE 3.6 Truth Table for Operator ||

p ₁	p ₂	p ₁ p ₂	Example (assume age = 24, weight = 140)
false	false	false	(age > 34) (weight >= 150) is false, because (age > 34) and (weight >= 150) are both false.
false	true	true	
true	false	true	(age > 18) (weight < 140) is true, because (age > 18) is true.
true	true	true	

The or (||) of two Boolean operands is **true** if at least one of the operands is **true**

Truth Table for Operator ^

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	

- The exclusive or (^) of two Boolean operands is **true** if and only if the two operands have different Boolean values.
- Note that **p1 ^ p2** is the same as **p1 != p2**.
- This could be thought as "A or B, but not, A and B".

Lazy operators

- If one of the operands of an **&&** operator is **false**, the expression is **false**; if one of the operands of an **||** operator is **true**, the expression is **true**.
- Java uses these properties to improve the performance of these operators.
- When evaluating **p1 && p2**, Java first evaluates **p1** and then, if **p1** is **true**, evaluates **p2**; if **p1** is **false**, it does not evaluate **p2**.
- When evaluating **p1 || p2**, Java first evaluates **p1** and then, if **p1** is **false**, evaluates **p2**; if **p1** is **true**, it does not evaluate **p2**.
- In programming language terminology, **&&** and **||** are known as the *short-circuit* or *lazy* operators.

Common error

- In mathematics, the expression $1 \leq \text{numberOfDaysInAMonth} \leq 31$ is correct.
- However, it is incorrect in Java, because $1 \leq \text{numberOfDaysInAMonth}$ is evaluated to a **boolean** value, which cannot be compared with **31**.
- Here, two operands (a **boolean** value and a numeric value) are *incompatible*. The correct expression in Java is
 $(1 \leq \text{numberOfDaysInAMonth}) \ \&\& \ (\text{numberOfDaysInAMonth} \leq 31)$

Logical Operators: an example to determine leap year

- A year is a leap year if it is divisible by 4 but not by 100, or if it is divisible by 400.
- You can use the following Boolean expressions to check whether a year is a leap year:

// A leap year is divisible by 4

boolean isLeapYear = (year % 4 == 0);

// A leap year is divisible by 4 but not by 100

isLeapYear = isLeapYear && (year % 100 != 0);

// A leap year is divisible by 4 but not by 100 or divisible by 400

isLeapYear = isLeapYear || (year % 400 == 0);

Or you can combine all these expressions into one like this:

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

LISTING 3.7 LeapYear.java

```
1 import java.util.Scanner;
2
3 public class LeapYear {
4     public static void main(String[] args) {
5         // Create a Scanner
6         Scanner input = new Scanner(System.in);
7         System.out.print("Enter a year: ");
8         int year = input.nextInt();
9
10        // Check if the year is a leap year
11        boolean isLeapYear =
12            (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
13
14        // Display the result
15        System.out.println(year + " is a leap year?" + isLeapYear);
16    }
17 }
```


Switch Statements vs if statements

- It is often the case that the value of a variable determines which branch a program should take:


```
if (x==1)
    statement;
else if (x==2)
    statement;
else
    statement;
```
- This is tedious!
- Java provides an alternative called the switch statement:


```
switch (x) {
    case 1: statement;
           break;
    case 2: statement;
           break;
    default: statement;
}
```

Programming I --- Ch. 3

33

Switch Statements

- A **switch** statement executes statements based on the value of a variable or an expression.
- The switch statement is a multi-way branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.
- The syntax for the **switch** 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 keyword **break** is optional. The **break** statement immediately ends the **switch** statement.
- Do not forget to use a **break** statement when one is needed.
- Once a case is matched, the statements starting from the matched case are executed until a break statement or the end of the **switch** statement is reached.
- This is referred to as *fall-through* behavior.

Programming I --- Ch. 3

34

Switch statement: *fall-through* behavior

Can you tell what does the following switch statement do?

```
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");
}
```

The code displays **Weekdays** for day of **1 to 5** and **Weekends** for day **0** and **6**.

- How to rewrite the above code with if-else statements?
- Note that 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.

Programming I --- Ch. 3

35

Switch Statements – equivalent if-else statement

```
switch (status) {
    case 0: compute tax for single filers;
            break;
    case 1: compute tax for married jointly or qualifying widow(er);
            break;
    case 2: compute tax for married filing separately;
            break;
    case 3: compute tax for head of household;
            break;
    default: System.out.println("Error: invalid status");
             System.exit(1);
}
```

System.exit(system call) terminates the currently running Java virtual machine by initiating its shutdown sequence. The argument serves as a **status code**. By convention, a nonzero status code indicates abnormal termination.

- System.exit(0); // EXIT_Everything Okay
- System.exit(1); // EXIT_FAILURE: Something expected goes wrong
- System.exit(-1); // EXIT_ERROR: Something unexpected goes wrong

The above switch statement checks whether the status matches the value **0**, **1**, **2**, or **3**, in that order. If matched, the corresponding tax is computed; if not matched, a message is displayed.

System.exit(1) is to terminate the program without executing any remaining statements left.

The equivalent if coded using multi-way if-else statement:

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

Programming I --- Ch. 3

36

Flowchart of the switch statement example

- The flowchart of the preceding example:

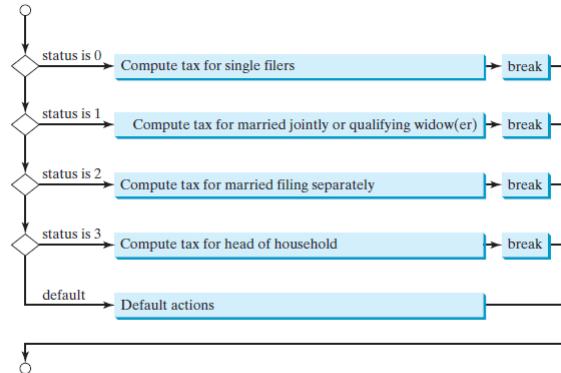


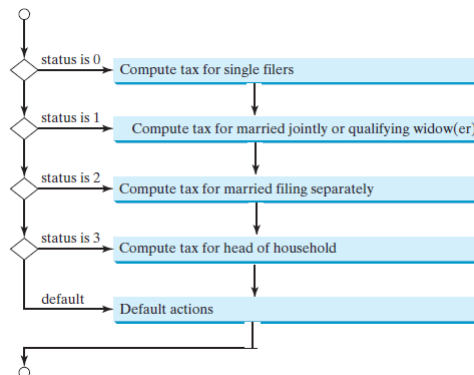
FIGURE 3.5 The **switch** statement checks all cases and executes the statements in the matched case.

Programming I --- Ch. 3

37

Flowchart of the switch statement example

- The flowchart of the preceding example **if the break statement is omitted (fall-through)**:



Programming I --- Ch. 3

38

The rules for switch statement

The **switch** statement observes the following rules:

- The **switch-expression** must yield a value of **char**, **byte**, **short**, **int**, or **String** type and must always be enclosed in parentheses. (The **char** and **String** types will be introduced in the next chapter.)
- The **value1**, ..., and **valueN** must have the same data type as the value of the **switch-expression**.
- Duplicate case values are not allowed.
- The case value must be a constant or a literal. Variables are not allowed.
- 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;
}
```

Programming I --- Ch. 3

39

Switch statement: an example

- Listing 3.9 finds out the Chinese Zodiac sign for a given year. The Chinese Zodiac is based on a twelve-year cycle, with each year represented by an animal.



FIGURE 3.6 The Chinese Zodiac is based on a twelve-year cycle.

- Imagine how the code will be written with if-else statements.
- If you have a lot of if checks, switch keeps your code looking cleaner, however that's really a matter of personal preference.

LISTING 3.9 ChineseZodiac.java

```
1 import java.util.Scanner;
2
3 public class ChineseZodiac {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6
7         System.out.print("Enter a year: ");
8         int year = input.nextInt();
9
10        switch (year % 12) {
11            case 0: System.out.println("monkey"); break;
12            case 1: System.out.println("rooster"); break;
13            case 2: System.out.println("dog"); break;
14            case 3: System.out.println("pig"); break;
15            case 4: System.out.println("rat"); break;
16            case 5: System.out.println("ox"); break;
17            case 6: System.out.println("tiger"); break;
18            case 7: System.out.println("rabbit"); break;
19            case 8: System.out.println("dragon"); break;
20            case 9: System.out.println("snake"); break;
21            case 10: System.out.println("horse"); break;
22            case 11: System.out.println("sheep"); break;
23        }
24    }
25 }
```

Programming I --- Ch. 3

40

Nested Switch statement

```
String Branch = "CSE";
int year = 2;
```

```
switch (year) {
case 1:
    System.out.println("elective courses : Advance english, Algebra");
    break;
case 2:
    switch (Branch) // nested switch
    {
        case "CSE":
        case "ECE":
            System.out.println("elective courses : Machine Learning, Big Data");
            break;

        case "ECE":
            System.out.println("elective courses : Antenna Engineering");
            break;

        default:
            System.out.println("Elective courses : Optimization");
    }
}
```

Programming I --- Ch. 3

41

Switch statement vs if-else statement

- Every switch statement has an equivalent if-else-if statement, but not vice versa.
- Switch statement is used when there is one expression that gets evaluated to a value and that value can be one of predefined set of values.
- If you need to perform multiple boolean / comparison operations at run-time, then if-else-if needs to be used.
- Besides, remember that we mentioned that the **switch-expression** must yield a value of **char**, **byte**, **short**, **int**, or **String** type. That means if you want to test the condition of GPA > 1.5, you need an if statement.

Programming I --- Ch. 3

42

Operator Precedence and Associativity

- *Operator precedence and associativity determine the order in which operators are evaluated.* Precedence solves ambiguities.
- Section 2.11 already introduced operator precedence involving arithmetic operators.
 - Operators contained within pairs of parentheses are evaluated first.
 - Parentheses can be nested, in which case the expression in the inner parentheses is evaluated first.
- This section discusses operator precedence in more details.
- 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*.

TABLE 3.8 Operator Precedence Chart

Precedence	Operator
	<code>var++</code> and <code>var--</code> (Postfix)
	<code>+</code> , <code>-</code> (Unary plus and minus), <code>++var</code> and <code>--var</code> (Prefix)
	(type) (Casting)
	<code>!</code> (Not)
	<code>*</code> , <code>/</code> , <code>%</code> (Multiplication, division, and remainder)
	<code>+</code> , <code>-</code> (Binary addition and subtraction)
	<code><</code> , <code><=</code> , <code>></code> , <code>>=</code> (Relational)
	<code>==</code> , <code>!=</code> (Equality)
	<code>^</code> (Exclusive OR)
	<code>&&</code> (AND)
	<code> </code> (OR)
	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code> (Assignment operator)

Operator Precedence and Associativity: examples

- For example, since `+` and `-` are of the same precedence and are left associative, the expression `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))`

Suppose **a**, **b**, and **c** are **1** before the assignment; after the whole expression is evaluated, **a** becomes **6**, **b** becomes **6**, and **c** becomes **5**.

- Suppose that you have this expression:

`3 + 4 * 4 > 5 * (4 + 3) - 1`

- What is its value? What is the execution order of the operators?

Example

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

$3 + 4 * 4 > 5 * (4 + 3) - 1$
 \uparrow (1) inside parentheses first
 $3 + 4 * 4 > 5 * 7 - 1$
 \uparrow (2) multiplication
 $3 + 16 > 5 * 7 - 1$
 \uparrow (3) multiplication
 $3 + 16 > 35 - 1$
 \uparrow (4) addition
 $19 > 35 - 1$
 \uparrow (5) subtraction
 $19 > 34$
 \uparrow (6) greater than
 false

Programming I --- Ch. 3

45

Debugging

- Debugging is the process of finding and fixing errors in a program.*
- Syntax errors are easy to find and easy to correct because the compiler gives indications as to where the errors came from and why they are there.
- Runtime errors are not difficult to find either, because the Java interpreter displays them on the console when the program aborts.
- Finding logic errors, on the other hand, can be very challenging.
- Logic errors are called *bugs*. The process of finding and correcting errors is called *debugging*.

Programming I --- Ch. 3

46

Debugging methods

- 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.
- These approaches might work for debugging a short, simple program, but for a large, complex program, the most effective approach is to use a debugger utility.

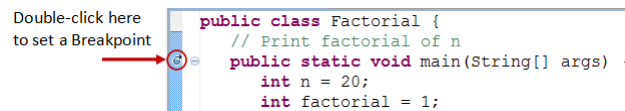
Debugger utilities

- The debugger utilities let you follow the execution of a program. They vary from one system to another, but they all support most of the following helpful features.
 - **Executing a single statement at a time** (*so that you can see the effect of each statement*)
 - **Tracing into or stepping over a method**
 - **Setting breakpoints**
 - Your program pauses when it reaches a breakpoint. Breakpoints are particularly useful when you know where your programming error starts. You can set a breakpoint at that statement and have the program execute until it reaches the breakpoint
 - **Displaying variables**
 - As you trace through a program, the content of a variable is continuously updated and displayed.
 - **Displaying call stacks**
 - **Modifying variables**
 - This is convenient when you want to test a program with different samples but do not want to leave the debugger.

Debugging Programs in Eclipse: Breakpoint

Step 1: Set an Initial Breakpoint

- A *breakpoint* suspends program execution for you to examine the internal states (e.g., value of variables) of the program. Before starting the debugger, you need to set at least one breakpoint to suspend the execution inside the program.
- Set a breakpoint at `main()` method by double-clicking on the *left-margin* of the line containing `main()`, or select "Toggle Breakpoint" from "Run" menu. A *blue circle* appears in the left-margin indicating a breakpoint is set at that line.



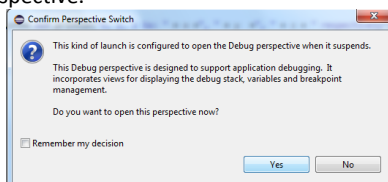
Programming I --- Ch. 3

49

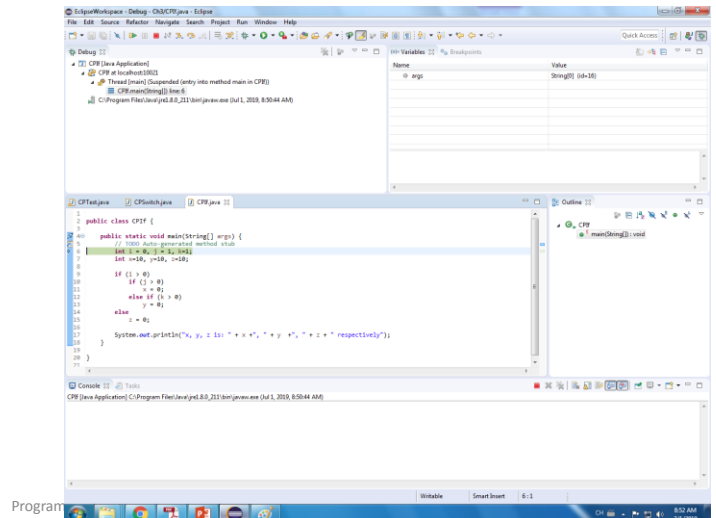
Debugging Programs in Eclipse: Breakpoint

Step 2: Start Debugger

- Right click anywhere on the source code (or from the "Run" menu) ⇒ "Debug As" ⇒ "Java Application" ⇒ choose "Yes" to switch into "Debug" perspective. A dialog box will appear to confirm to switch to Debug perspective.



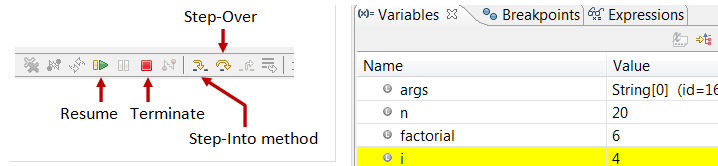
- A *perspective* is a particular arrangement of panels to suits a certain development task such as editing or debugging.
- The program begins execution but suspends its operation at the breakpoint, i.e., the `main()` method.



Debugging Programs in Eclipse: Breakpoint

Step 3: Step-Over and Watch the Variables and Outputs

- Click the "Step Over" button (or select "Step Over" from "Run" menu) to *single-step* through your program. At each of the step, examine the value of the variables (in the "Variable" panel) and the outputs produced by your program (in the "Console" Panel), if any. You can also place your cursor at any variable to inspect the content of the variable.



For **Step-Over**, the debugger executes one line of the program, *without stepping inside method calls*. Contrast that with the option **Step-Into**, which traces inside method calls.

Programming I --- Ch. 3


51

Debugging Programs in Eclipse: Breakpoint

Step 4: Breakpoint, Run-To-Line, Resume and Terminate

- "Resume" continues the program execution, up to the next breakpoint, or till the end of the program.
- Alternatively, you can place the cursor on a particular statement, and issue "Run-To-Line" from the "Run" menu to continue execution up to the line.
- "Terminate" ends the debugging session. Always terminate your current debugging session using "Terminate" or "Resume" till the end of the program.

Step 5: Switching Back to Java perspective

- Click the "Java" perspective icon  on the upper-right corner to switch back to the "Java" perspective for further programming (or "Window" menu ⇒ Open Perspective ⇒ Java).

Programming I --- Ch. 3

52

Debugging Programs in Eclipse: other features

- **Step-Into and Step-Return:** To debug a *method*, you need to use "Step-Into" to step into the *first* statement of the method. ("Step-Over" runs the function in a single step without stepping through the statements within the function.) You could use "Step-Return" to return back to the caller, anywhere within the method. Alternatively, you could set a breakpoint inside a method.
- **Modify the Value of a Variable:** You can modify the value of a variable by entering a new value in the "Variable" panel. This is handy for temporarily modifying the behavior of a program, without changing the source code.

Programming I --- Ch. 3

53

Chapter Summary

- A **boolean** type variable can store a **true** or **false** value.
- The relational operators (<, <=, ==, !=, >, >=) yield a Boolean value.
- *Selection statements* are used for programming with alternative courses of actions. There are several types of selection statements:
 - one-way **if** statements,
 - two-way **if-else** statements,
 - nested **if** statements,
 - multi-way **if-else** statements,
 - **switch** statements.
- The Boolean operators **&&**, **||**, **!**, and **^** operate with Boolean values and variables.
- The **switch** statement makes control decisions based on a switch expression of type **char**, **byte**, **short**, **int**, or **String**
- The operators in expressions are evaluated in the order determined by the rules of parentheses, *operator precedence*, and *operator associativity*.

Programming I --- Ch. 3

54

```

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

```

ses

- Write the output of the following code if there is any.

```

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

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

```

```

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

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

```

- Which of the following statements are equivalent?

```

if (i > 0) {
    if (j > 0)
        x = 0;
    else if (k > 0)
        y = 0;
}
else
    z = 0;

```

(a)

```

if (i > 0)
    if (j > 0)
        x = 0;
    else if (k > 0)
        y = 0;
else
    z = 0;

```

(b)

```

if (i > 0)
    if (j > 0)
        x = 0;
    else if (k > 0)
        y = 0;
else
    z = 0;

```

(c)

Programming I --- Ch. 3

55

Exercises

- Are the following statements equivalent? If YES, which one is better and explain?

```

if (age < 16)
    System.out.println(
        "Cannot get a driver's license");
if (age >= 16)
    System.out.println(
        "Can get a driver's license");

```

(a)

```

if (age < 16)
    System.out.println(
        "Cannot get a driver's license");
else
    System.out.println(
        "Can get a driver's license");

```

(b)

```

if (number % 2 == 0)
    System.out.println(
        (number + " is even"));
if (number % 5 == 0)
    System.out.println(
        (number + " is multiple of 5"));

```

(a)

```

if (number % 2 == 0)
    System.out.println(
        (number + " is even"));
else if (number % 5 == 0)
    System.out.println(
        (number + " is multiple of 5"));

```

(b)

- Draw flowcharts for the code above.

Programming I --- Ch. 3

56