Chapter 3

Selections

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 write expressions using the conditional expression
- To examine the rules governing operator precedence and associativity
- To apply common techniques to debug errors

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);
}
```

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*.

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

```java
import java.util.Scanner;

public class AdditionQuiz {
  public static void main(String[] args) {
    int number1 = (int)(System.currentTimeMillis() % 10);        generate number1
    int number2 = (int)(System.currentTimeMillis() / 7 % 10);    generate number2

    // Create a Scanner
    Scanner input = new Scanner(System.in);

    System.out.print(                                            show question
      "What is " + number1 + " + " + number2 + "? ");

    int answer = input.nextInt();

    System.out.println(                                          display result
      number1 + " + " + number2 + " = " + answer + " is " +
      (number1 + number2 == answer));
  }
}
```

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);
}
```

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

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

(a) Wrong

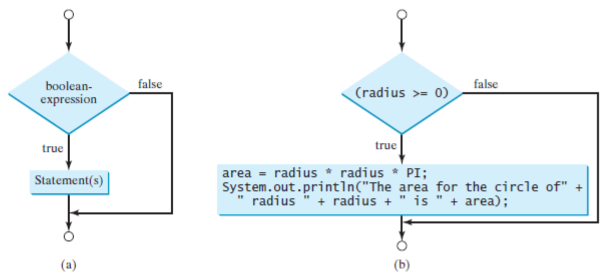
```
if (i > 0) {
  System.out.println("i is positive");
}
```

(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.

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.



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

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

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;
}
```

- 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.");
```

Flowchart of Two-way if-else Statements

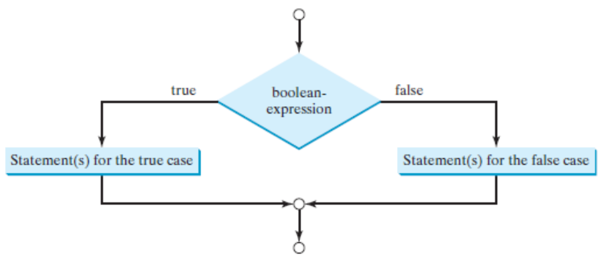

true
boolean-expression
false
Statement(s) for the true case
Statement(s) for the false case

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.

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.

```
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)

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

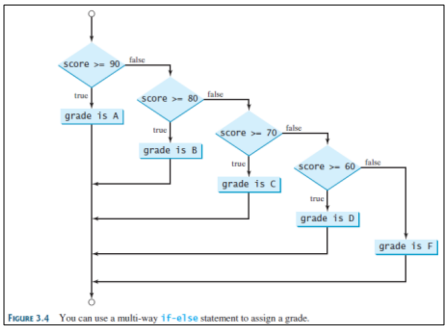


FIGURE 3.4 You can use a multi-way `if-else` statement to assign a grade.

Common Errors and Pitfalls using if statement

- **Redundant Testing of Boolean Values**

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

(a)

Equivalent

This is better

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

(b)

- **Simplifying Boolean Variable Assignment**

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

(a)

Equivalent

This is shorter

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

(b)

- **Avoiding Duplicate Code in Different Cases**

```
if (inState) {
  tuition = 5000;
  System.out.println("The tuition is " + tuition);
}
else {
  tuition = 15000;
  System.out.println("The tuition is " + tuition);
}
```

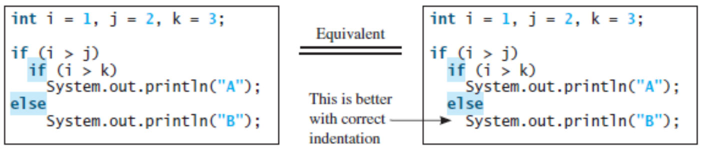
Equivalent

This is better

```
if (inState) {
  tuition = 5000;
}
else {
  tuition = 15000;
}
System.out.println("The tuition is " + tuition);
```

Common Errors and Pitfalls using if statement

- **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");
```

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

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

Equivalent
This is better with correct indentation

(a) (b)

  - This situation is known as the *dangling else ambiguity*. The **else** clause always matches the most recent unmatched **if** clause in the same block. So, the statement in (a) is equivalent to the code in (b).

Generating Random Numbers

- *You can use* **Math.random()** *to obtain a random double value* **d** such that 0.0 <= *d* < 1.0
- Thus, **(int)(Math.random() * 10)** returns a random single-digit integer (i.e., a number between **0** and **9**).
- 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

```java
import java.util.Scanner;

public class SubtractionQuiz {
  public static void main(String[] args) {
    // 1. Generate two random single-digit integers
    int number1 = (int)(Math.random() * 10);
    int number2 = (int)(Math.random() * 10);

    // 2. If number1 < number2, swap number1 with number2
    if (number1 < number2) {
      int temp = number1;
      number1 = number2;
      number2 = temp;
    }

    // 3. Prompt the student to answer "What is number1 – number2?"
    System.out.print
      ("What is " + number1 + " - " + number2 + "? ");
    Scanner input = new Scanner(System.in);
    int answer = input.nextInt();

    // 4. Grade the answer and display the result
    if (number1 - number2 == answer)
      System.out.println("You are correct!");
    else {
      System.out.println("Your answer is wrong.");
      System.out.println(number1 + " - " + number2 +
        " should be " + (number1 - number2));
    }
  }
}
```

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.

TABLE 3.3 Boolean Operators

Operator	*Name*	*Description*	
!	not	logical negation	negates **true** to **false** and **false** to **true**
&&	and	logical conjunction	The and (**&&**) of two Boolean operands is **true** if and only if both operands are **true**.
||	or	logical disjunction	The or (**||**) of two Boolean operands is **true** if at least one of the operands is **true**.
^	exclusive or	logical exclusion	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**.

- In mathematics, the expression **1** <= numberOfDaysInAMonth <= **31** is correct.
- However, it is incorrect in Java, because **1 <= 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** <= numberOfDaysInAMonth) && (numberOfDaysInAMonth <= **31**)

Truth Table

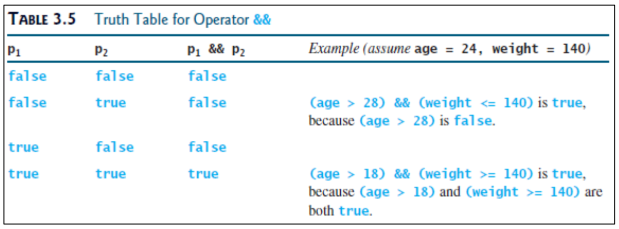


TABLE 3.5 Truth Table for Operator &&

p1	p2	p1 && p2	*Example (assume* age = 24, weight = 140*)*
false	false	false	
false	true	false	(age > 28) && (weight <= 140) is true, 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.

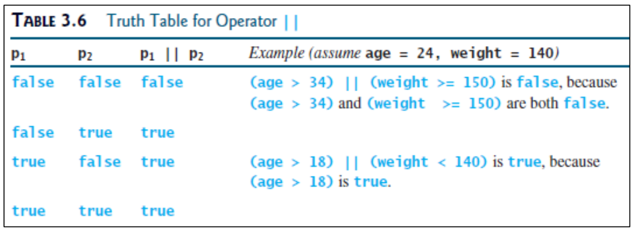


TABLE 3.6 Truth Table for Operator ||

p1	p2	p1 || p2	*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	

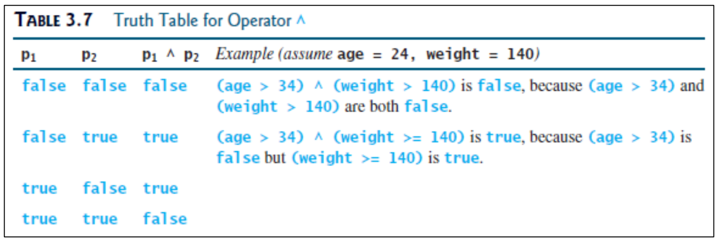


TABLE 3.7 Truth Table for Operator ^

p1	p2	p1 ^ p2	*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	

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.

De Morgan's law

- De Morgan's law, named after Indian-born British mathematician and logician Augustus De Morgan (1806–1871), can be used to simplify Boolean expressions. The law states:
  - !(condition1 && condition2) is the same as !condition1 || !condition2
  - !(condition1 || condition2) is the same as !condition1 && !condition2
- For example, ! (number % **2** == **0** && number % **3** == **0**) can be simplified using an equivalent expression:

  (number % **2** != **0** || number % **3** != **0**)
- As another example, !(number == **2** || number == **3**) is better written as number != **2** && number != **3**

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

```java
import java.util.Scanner;

public class LeapYear {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a year: ");
    int year = input.nextInt();

    // Check if the year is a leap year
    boolean isLeapYear =
      (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);

    // Display the result
    System.out.println(year + " is a leap year? " + isLeapYear);
  }
}
```

Switch Statements

- *A **switch** statement executes statements based on the value of a variable or an 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.

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

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);
}
```

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.

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 {
}
```

Flowchart of the switch statement example

- The flowchart of the preceding example:

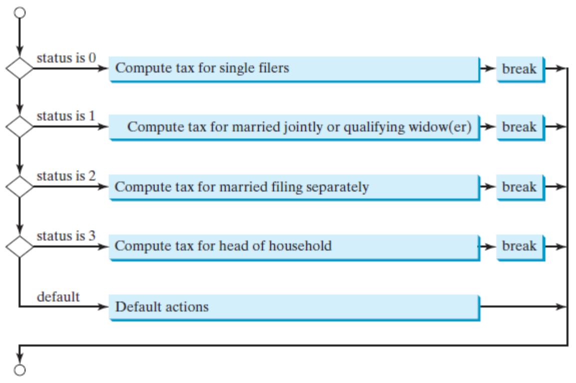

status is 0
Compute tax for single filers
break
status is 1
Compute tax for married jointly or qualifying widow(er)
break
status is 2
Compute tax for married filing separately
break
status is 3
Compute tax for head of household
break
default
Default actions

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

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");
}
```

- 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.

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.
- Imagine how the code will be written with if-else statements.

LISTING 3.9 ChineseZodiac.java

```
import java.util.Scanner;

public class ChineseZodiac {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    System.out.print("Enter a year: ");
    int year = input.nextInt();

    switch (year % 12) {
      case 0: System.out.println("monkey"); break;
      case 1: System.out.println("rooster"); break;
      case 2: System.out.println("dog"); break;
      case 3: System.out.println("pig"); break;
      case 4: System.out.println("rat"); break;
      case 5: System.out.println("ox"); break;
      case 6: System.out.println("tiger"); break;
      case 7: System.out.println("rabbit"); break;
      case 8: System.out.println("dragon"); break;
      case 9: System.out.println("snake"); break;
      case 10: System.out.println("horse"); break;
      case 11: System.out.println("sheep");
    }
  }
}
```

Conditional Expressions

- *A conditional expression evaluates an expression based on a condition.* The syntax is: `boolean-expression ? expression1 : expression2;`
- The result of this conditional expression is **expression1** if **boolean-expression** is true; otherwise the result is **expression2**.
- Suppose you want to assign the larger number of variable **num1** and **num2** to **max**. You can simply write a statement using the conditional expression: `max = (num1 > num2) ? num1 : num2;`
- Compare the above conditional expression with its equivalent if-else statement.

Operator Precedence and Associativity

- *Operator precedence and associativity determine the order in which operators are evaluated.*
- Section 2.11 already introduced operator precedence involving arithmetic operators (refer to next slide for recap).
- This section discusses operator precedence in more detail.
- 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*.

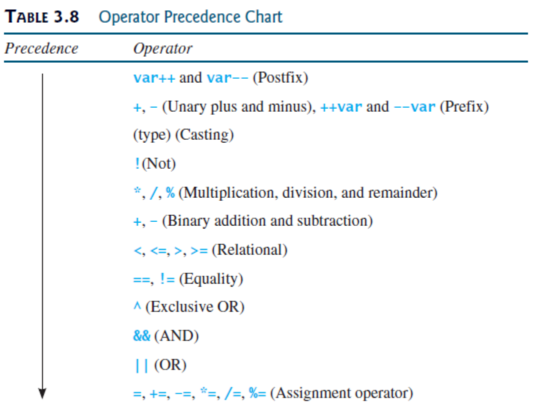


TABLE 3.8 Operator Precedence Chart

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

Evaluating expressions and operator precedence (recap from Ch. 2)

- 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.
- When more than one operator is used in an expression, the following operator precedence rule is used to determine the order of evaluation.
  - Multiplication, division, and remainder operators are applied first. If an expression contains several multiplication, division, and remainder operators, they are applied from left to right.
  - Addition and subtraction operators are applied last. If an expression contains several addition and subtraction operators, they are applied from left to right.

Operator Precedence and Associativity: examples

- For example, since **+** and **–** are of the same precedence and are left associative, the expression `a - b + c - d` $\xlongequal{\text{is equivalent to}}$ `((a - b) + c) - d`
- Assignment operators are *right associative*. Therefore, the expression

  `a = b += c = 5` $\xlongequal{\text{is equivalent to}}$ `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**.

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*.

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.

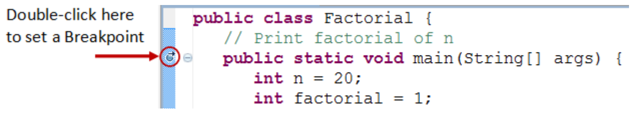
Debugger utilities

- 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.

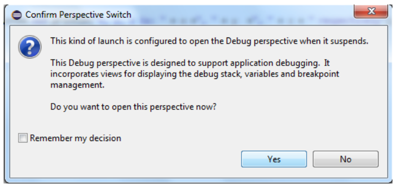


```
public class Factorial {
   // Print factorial of n
   public static void main(String[] args) {
      int n = 20;
      int factorial = 1;
```

Debugging Programs in Eclipse: Breakpoint

- **Step 2: Start Debugger**

  A dialog box will appear to confirm to switch to Debug perspective.



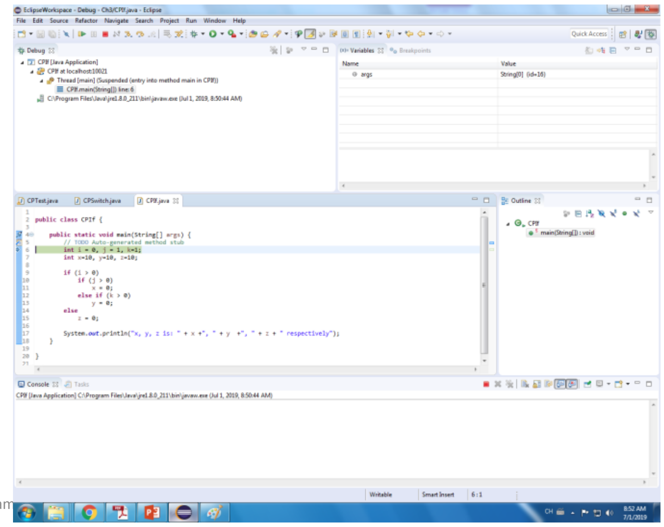
Confirm Perspective Switch

This kind of launch is configured to open the Debug perspective when it suspends.

This Debug perspective is designed to support application debugging. It incorporates views for displaying the debug stack, variables and breakpoint management.

Do you want to open this perspective now?

Remember my decision



EclipseWorkspace - Debug - Ch3/CPIf.java - Eclipse

```
CPIf [Java Application]
  CPIf at localhost:10021
    Thread [main] (Suspended (entry into method main in CPIf))
      CPIf.main(String[]) line: 6
  C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (Jul 1, 2019, 8:50:44 AM)
```

Name	Value
args	String[0] (id=16)

```java
public class CPIf {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int i = 0, j = 1, k=1;
        int x=10, y=10, z=10;

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

        System.out.println("x, y, z is: " + x +", " + y  +", " + z + " respectively");
    }

}
```

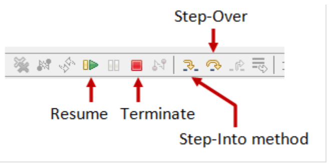
Outline: CPIf — main(String[]) : void

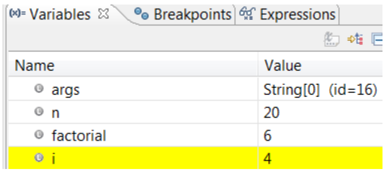
CPIf [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (Jul 1, 2019, 8:50:44 AM)

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.





Name	Value
args	String[0] (id=16)
n	20
factorial	6
i	4

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.

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).

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.

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, and
  - conditional expressions.
- 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*.

Ideas for further practice

- (*Random month*) Write a program that randomly generates an integer between 1 and 12 and displays the English month name January, February, …, December for the number 1, 2, …, 12, accordingly.
- (*Find future dates*) Write a program that prompts the user to enter an integer for today's day of the week (Sunday is 0, Monday is 1, …, and Saturday is 6). Also prompt the user to enter the number of days after today for a future day and display the future day of the week. Here is a sample run:

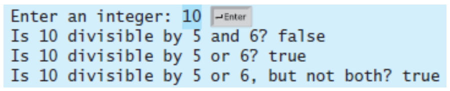
```
Enter today's day: 1 ↵Enter
Enter the number of days elapsed since today: 3 ↵Enter
Today is Monday and the future day is Thursday
```

- (*Find the number of days in a month*) Write a program that prompts the user to enter the month and year and displays the number of days in the month. For example, if the user entered month **2** and year **2012**, the program should display that February 2012 had 29 days. If the user entered month **3** and year **2015**, the program should display that March 2015 had 31 days.
- (*Game: pick a card*) Write a program that simulates picking a card from a deck of **52** cards. Your program should display the rank (**Ace**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **10**, **Jack**, **Queen**, **King**) and suit (**Clubs**, **Diamonds**, **Hearts**, **Spades**) of the card. Here is a sample run of the program:

```
The card you picked is Jack of Hearts
```

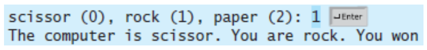
Ideas for further practice

- (*Use the* **&&**, **||** *and* **^** *operators*) Write a program that prompts the user to enter an integer and determines whether it is divisible by 5 and 6, whether it is divisible by 5 or 6, and whether it is divisible by 5 or 6, but not both. Here is a sample run of this program:

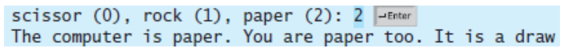


```
Enter an integer: 10
Is 10 divisible by 5 and 6? false
Is 10 divisible by 5 or 6? true
Is 10 divisible by 5 or 6, but not both? true
```

- (*Game: scissor, rock, paper*) Write a program that plays the popular scissor-rock-paper game. The program randomly generates a number **0**, **1**, or **2** representing scissor, rock, and paper. The program prompts the user to enter a number **0**, **1**, or **2** and displays a message indicating whether the user or the computer wins, loses, or draws. Here are sample runs:



```
scissor (0), rock (1), paper (2): 1
The computer is scissor. You are rock. You won
```



```
scissor (0), rock (1), paper (2): 2
The computer is paper. You are paper too. It is a draw
```