Java Programming, 9e

Chapter 5

Making Decisions

CENGAGE

Objectives (1 of 2)

- Plan decision-making logic
- Make decisions with the `if` and `if…else` statements
- Use multiple statements in `if` and `if…else` clauses
- Nest `if` and `if…else` statements
- Use AND and OR operators

Objectives (2 of 2)

- Make accurate and efficient decisions
- Use the `switch` statement
- Use the conditional and NOT operators
- Assess operator precedence
- Add decisions and constructors to instance methods

Planning Decision-Making Logic (1 of 5)

- **Pseudocode**
  - Use paper and a pencil
  - Plan a program’s logic by writing plain English statements
  - Accomplish important steps in a given task
  - Use everyday language
- **Flowchart**
  - Steps in diagram form
  - A series of shapes connected by arrows

Planning Decision-Making Logic (2 of 5)

- **Flowchart** (cont'd.)
  - Programmers use a variety of shapes for different tasks
    - Rectangle to represent any unconditional step
    - Diamond to represent any decision
- **Sequence structure**
  - One step follows another unconditionally
  - Cannot branch away or skip a step

Planning Decision-Making Logic (3 of 5)

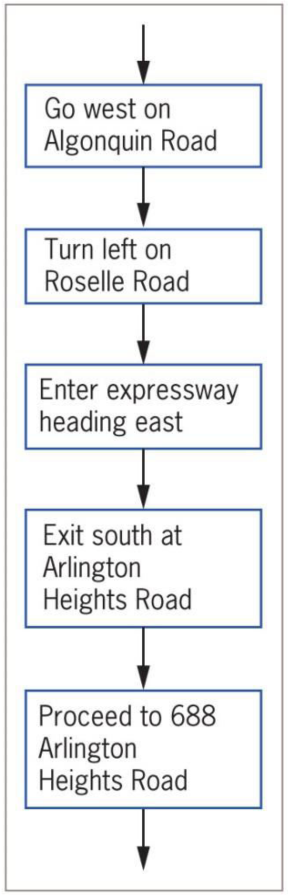

Go west on Algonquin Road
Turn left on Roselle Road
Enter expressway heading east
Exit south at Arlington Heights Road
Proceed to 688 Arlington Heights Road

Figure 5-1 Flowchart of a series of sequential steps

Planning Decision-Making Logic (4 of 5)

- **Decision structure**
  - Involves choosing among alternative courses of action
  - Based on some value within a program
- All computer decisions are yes-or-no decisions
- **Boolean values**
  - `true` and `false` values
  - Used in every computer decision

Planning Decision-Making Logic (5 of 5)

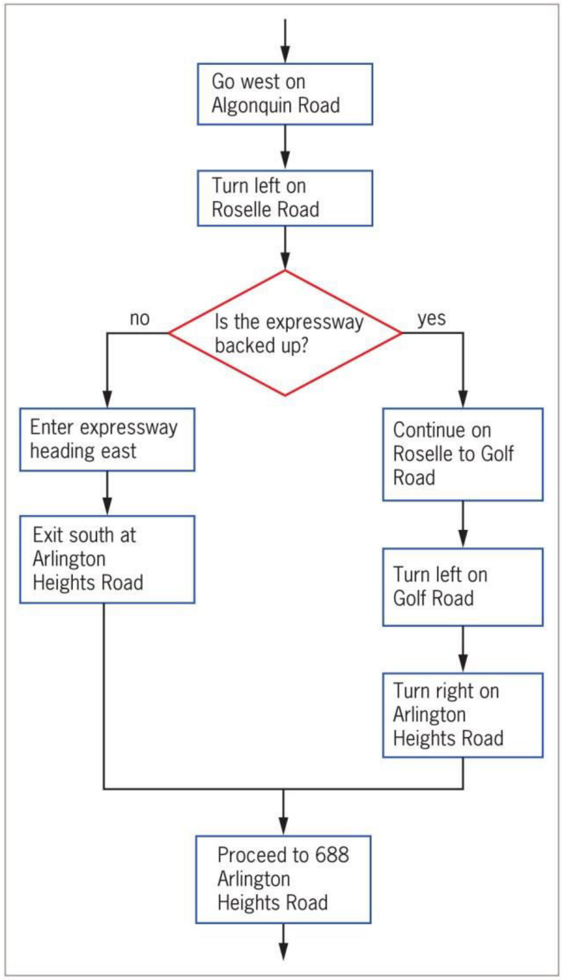
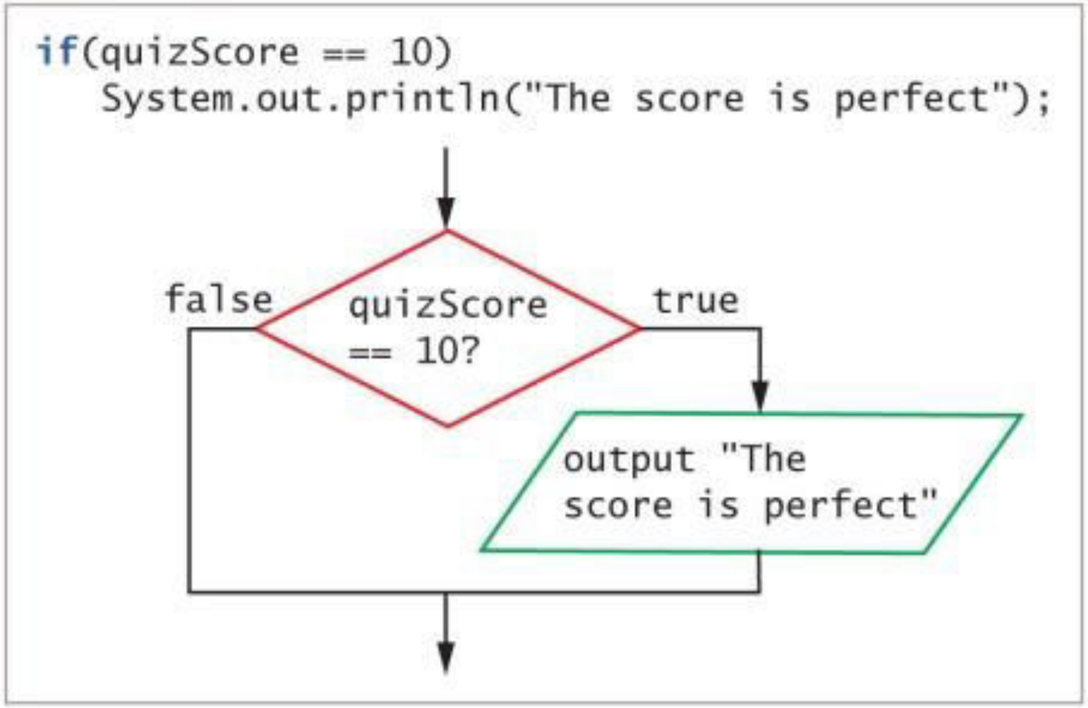

Go west on Algonquin Road
Turn left on Roselle Road
Is the expressway backed up?
no
yes
Enter expressway heading east
Exit south at Arlington Heights Road
Continue on Roselle to Golf Road
Turn left on Golf Road
Turn right on Arlington Heights Road
Proceed to 688 Arlington Heights Road

Figure 5-2 Flowchart including a decision

The `if` and `if…else` Statements (1 of 2)

- **`if` statement**
  - The simplest statement to make a decision
  - A Boolean expression appears within parentheses
  - No space between the keyword `if` and the opening parenthesis
  - Execution always continues to the next independent statement
  - Use a double equal sign (`==`) to determine equivalency

The `if` and `if…else` Statements (2 of 2)



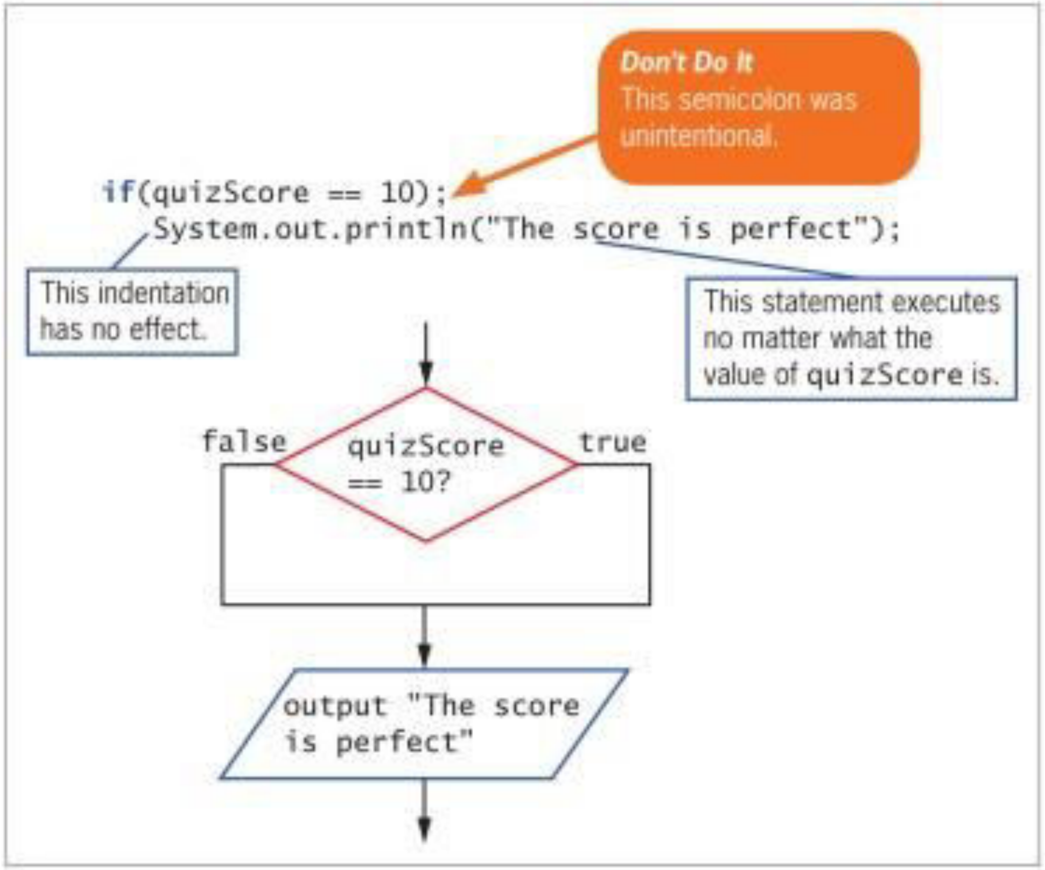
```
if(quizScore == 10)
   System.out.println("The score is perfect");
```

Figure 5-3 A Java `if` statement and its logic

Pitfall: Misplacing a Semicolon in an `if` Statement (1 of 2)

- There should be no semicolon at the end of the first line of the `if` statement
  - `if(someVariable == 10)`
  - The statement does not end there
- When a semicolon follows `if` directly:
  - An **empty statement** contains only a semicolon
  - Execution continues with the next independent statement

Pitfall: Misplacing a Semicolon in an `if` Statement (2 of 2)



```
if(quizScore == 10);
   System.out.println("The score is perfect");
```

Don't Do It
This semicolon was unintentional.
This indentation has no effect.
This statement executes no matter what the value of quizScore is.
false
quizScore == 10?
true
output "The score is perfect"

Figure 5-4 Logic that executes when an extra semicolon is inserted in an `if` statement

Pitfall: Using the Assignment Operator Instead of the Equivalency Operator

- Attempt to determine equivalency
  - Using a single equal sign rather than a double equal sign is illegal
- You can store a Boolean expression's value in a Boolean variable before using it in an `if` statement

Pitfall: Attempting to Compare Objects Using the Relational Operators

- Use standard relational operators to compare values of primitive data types
  - Not objects
- You can use the equals and not equals comparisons (== and !=) with objects
  - Compare objects’ memory addresses instead of values

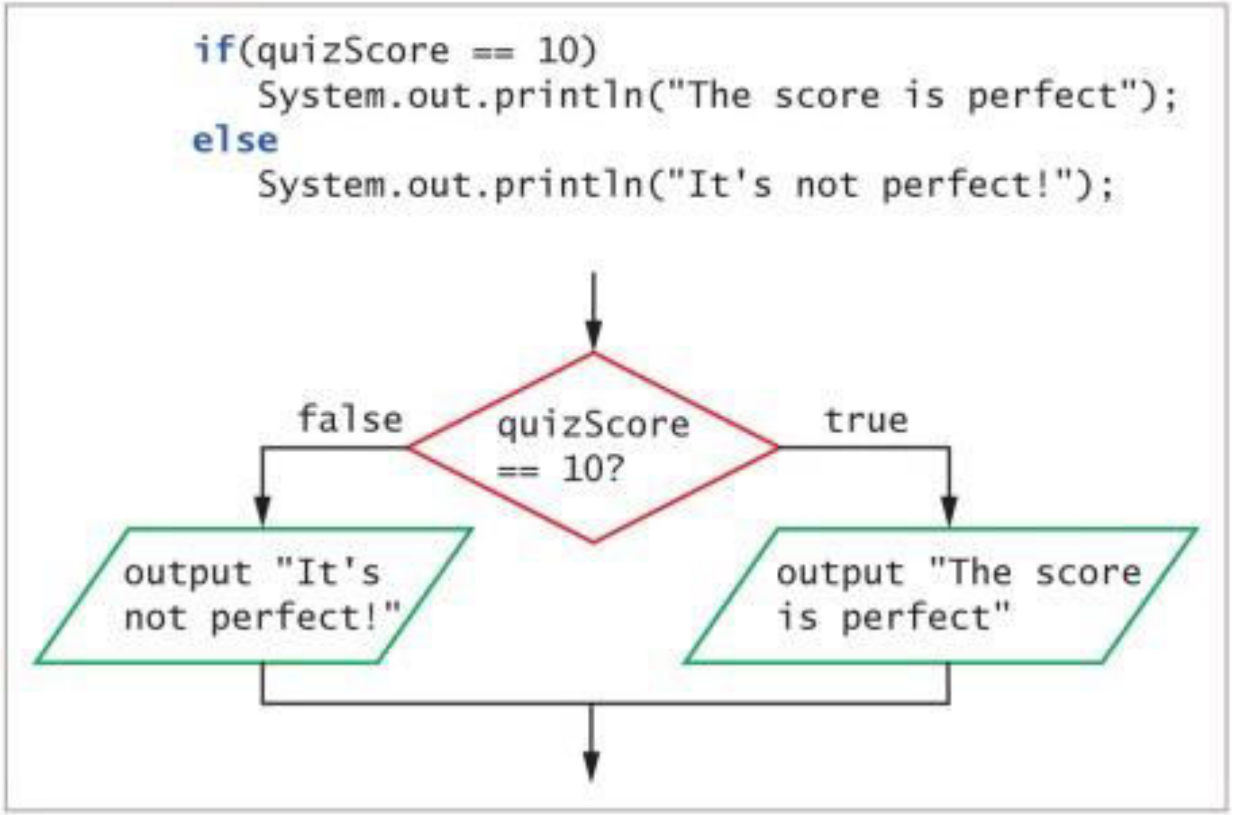
The `if…else` Statement (1 of 3)

- **Single-alternative `if`**
  - Perform an action, or not
    - Based on one alternative
- **Dual-alternative `if`**
  - Two possible courses of action
- **`if…else` statement**
  - Performs one action when a Boolean expression evaluates as `true`
  - Performs a different action when a Boolean expression evaluates as `false`

The `if`…`else` Statement (2 of 3)

- **`if`…`else` statement** (cont’d.)
  - A statement that executes when `if` is `true` or `false` and ends with a semicolon
  - Vertically align the keyword `if` with the keyword `else`
  - Illegal to code `else` without `if`
  - Depending on the evaluation of the Boolean expression following `if`, only one resulting action takes place

The `if…else` Statement (3 of 3)



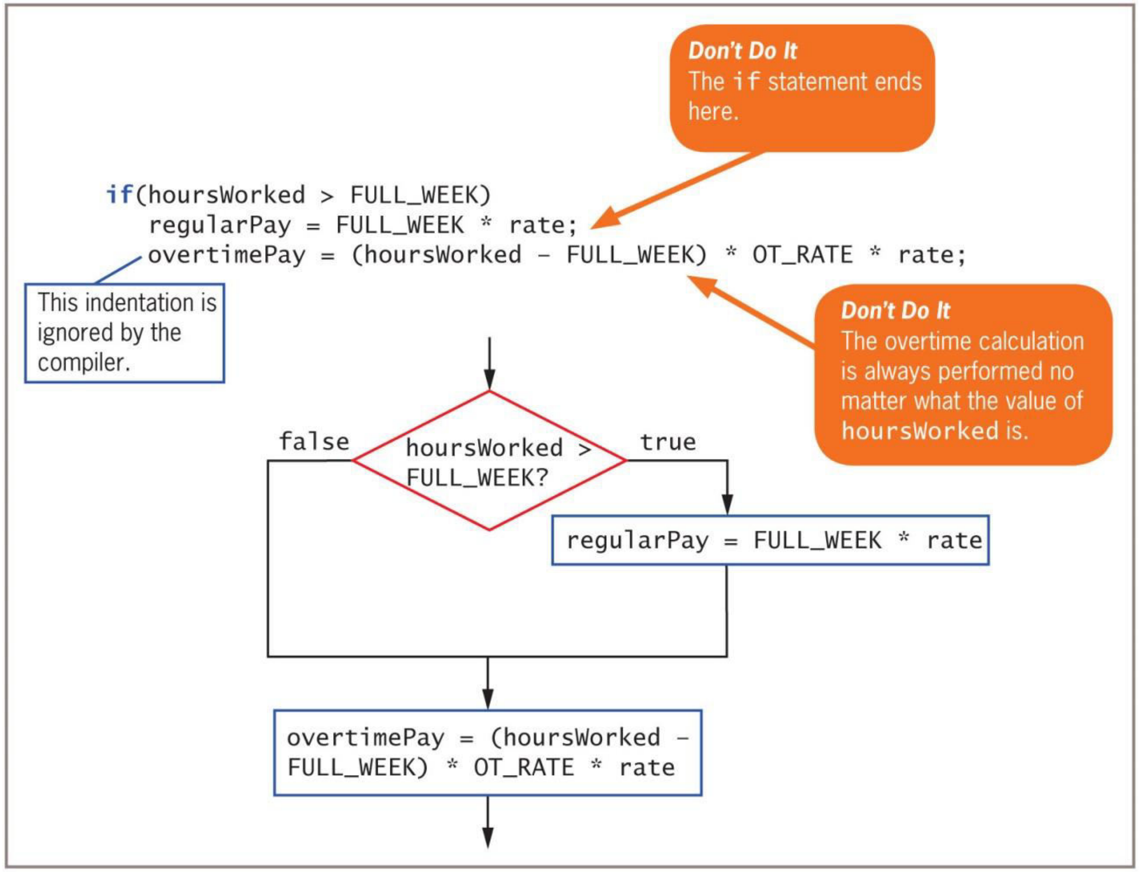
```
if(quizScore == 10)
   System.out.println("The score is perfect");
else
   System.out.println("It's not perfect!");
```

Figure 5-5 An `if…else` statement and its logic

Using Multiple Statements in `if` and `if…else` Clauses (1 of 2)

- To execute more than one statement, use a pair of curly braces
  - Place dependent statements within a block
  - Crucial to place the curly braces correctly
- Any variable declared within a block is local to that block

Using Multiple Statements in `if` and `if…else` Clauses (2 of 2)



```
if(hoursWorked > FULL_WEEK)
   regularPay = FULL_WEEK * rate;
   overtimePay = (hoursWorked - FULL_WEEK) * OT_RATE * rate;
```

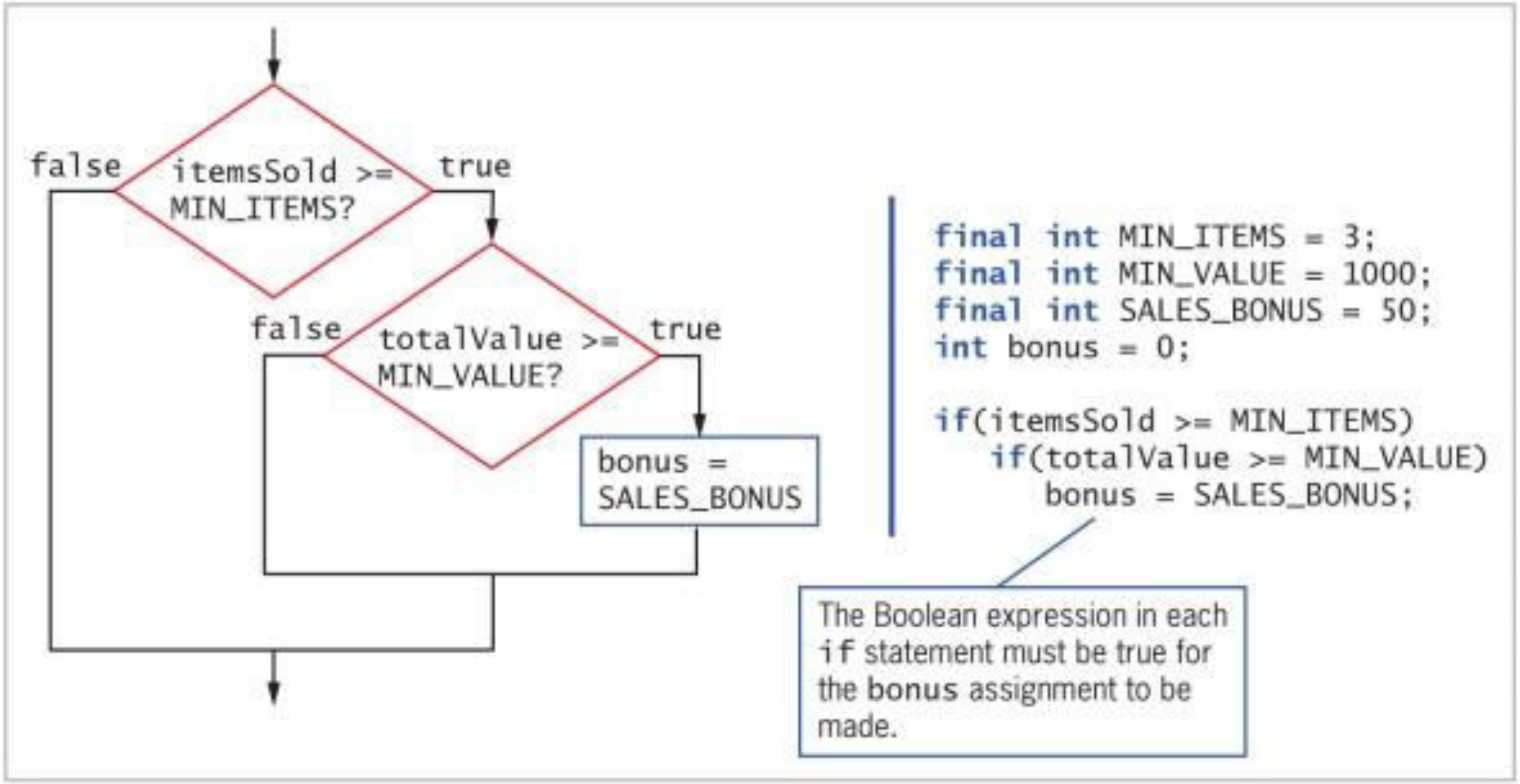
Don't Do It
The if statement ends here.
This indentation is ignored by the compiler.
Don't Do It
The overtime calculation is always performed no matter what the value of hoursWorked is.
false
hoursWorked > FULL_WEEK?
true
regularPay = FULL_WEEK * rate
overtimePay = (hoursWorked - FULL_WEEK) * OT_RATE * rate

Figure 5-8 Erroneous overtime pay calculation with missing curly braces

Nesting `if` and `if...else` Statements (1 of 2)

- **Nested `if` statements**
  - Statements in which an `if` structure is contained inside another `if` structure
  - Two conditions must be met before some action is taken
- Pay careful attention to the placement of `else` clauses
- `else` statements are always associated with `if` on a "first in-last out" basis

Nesting `if` and `if…else` Statements (2 of 2)



```
final int MIN_ITEMS = 3;
final int MIN_VALUE = 1000;
final int SALES_BONUS = 50;
int bonus = 0;

if(itemsSold >= MIN_ITEMS)
   if(totalValue >= MIN_VALUE)
      bonus = SALES_BONUS;
```

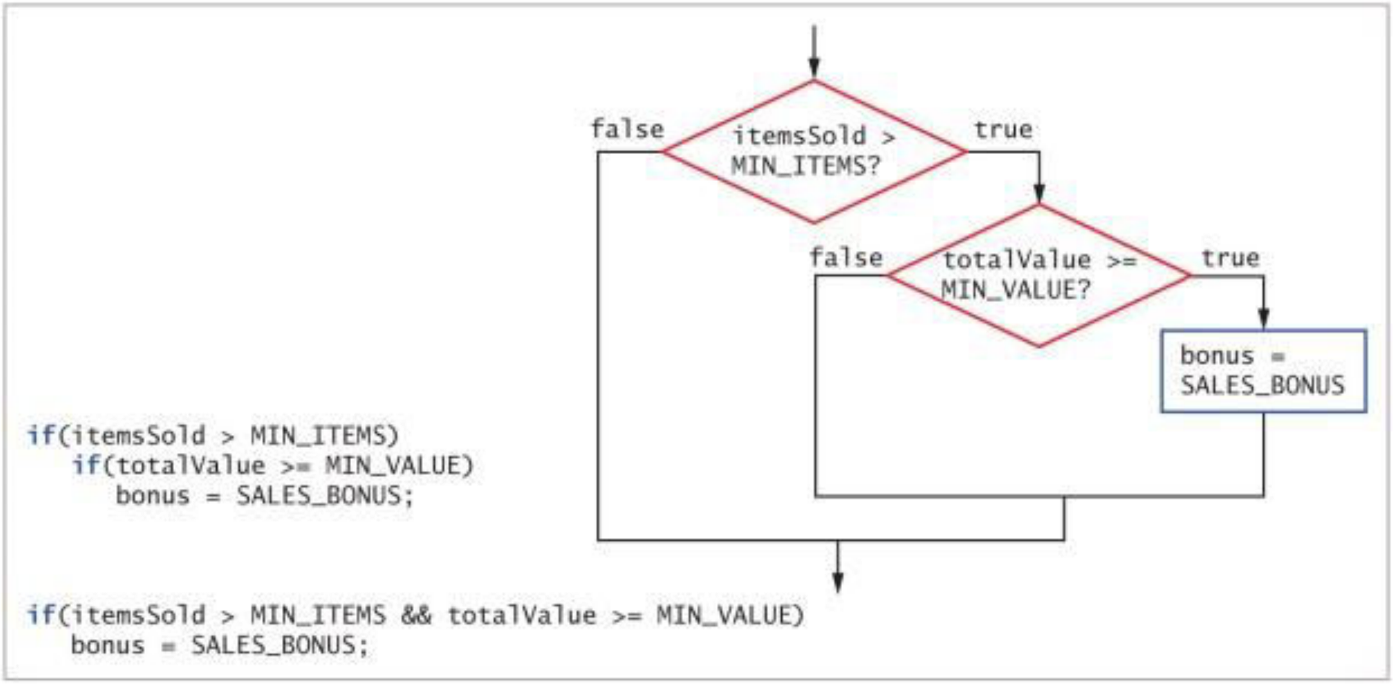
The Boolean expression in each `if` statement must be true for the bonus assignment to be made.

Figure 5-12 Determining whether to assign a bonus using nested `if` statements

Using Logical AND and OR Operators (1 of 5)

- The **logical AND operator**
  - An alternative to some nested `if` statements
  - Used between two Boolean expressions to determine whether both are `true`
  - Written as two ampersands (`&&`)
    - Include a complete Boolean expression on each side
  - Both Boolean expressions that surround the operator must be true before the action in the statement can occur

Using Logical AND and OR Operators (2 of 5)



```
if(itemsSold > MIN_ITEMS)
   if(totalValue >= MIN_VALUE)
      bonus = SALES_BONUS;
```

```
if(itemsSold > MIN_ITEMS && totalValue >= MIN_VALUE)
   bonus = SALES_BONUS;
```

Figure 5-15 Code and logic for bonus-determining decision using nested ifs and the && operator

Using Logical AND and OR Operators (3 of 5)

- The **OR operator**
  - An action to occur when at least one of two conditions is true
  - Written as | |
    - Sometimes called pipes

Using Logical AND and OR Operators (4 of 5)

```
if(itemsBought >= MIN_ITEMS)
   discountRate = DISCOUNT;
else
    if(itemsValue >= MIN_VALUE)
       discountRate = DISCOUNT;

if(itemsBought >= MIN_ITEMS || itemsValue >= MIN_VALUE)
   discountRate = DISCOUNT;
```

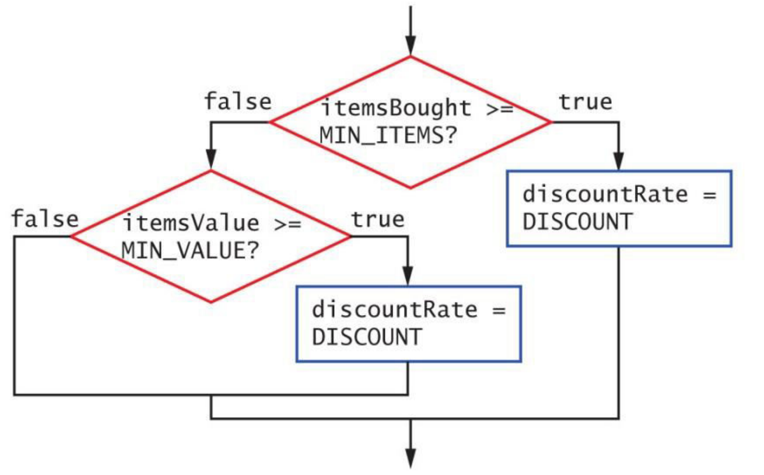


Figure 5-16 Determining customer discount when customer needs to meet only one of two criteria

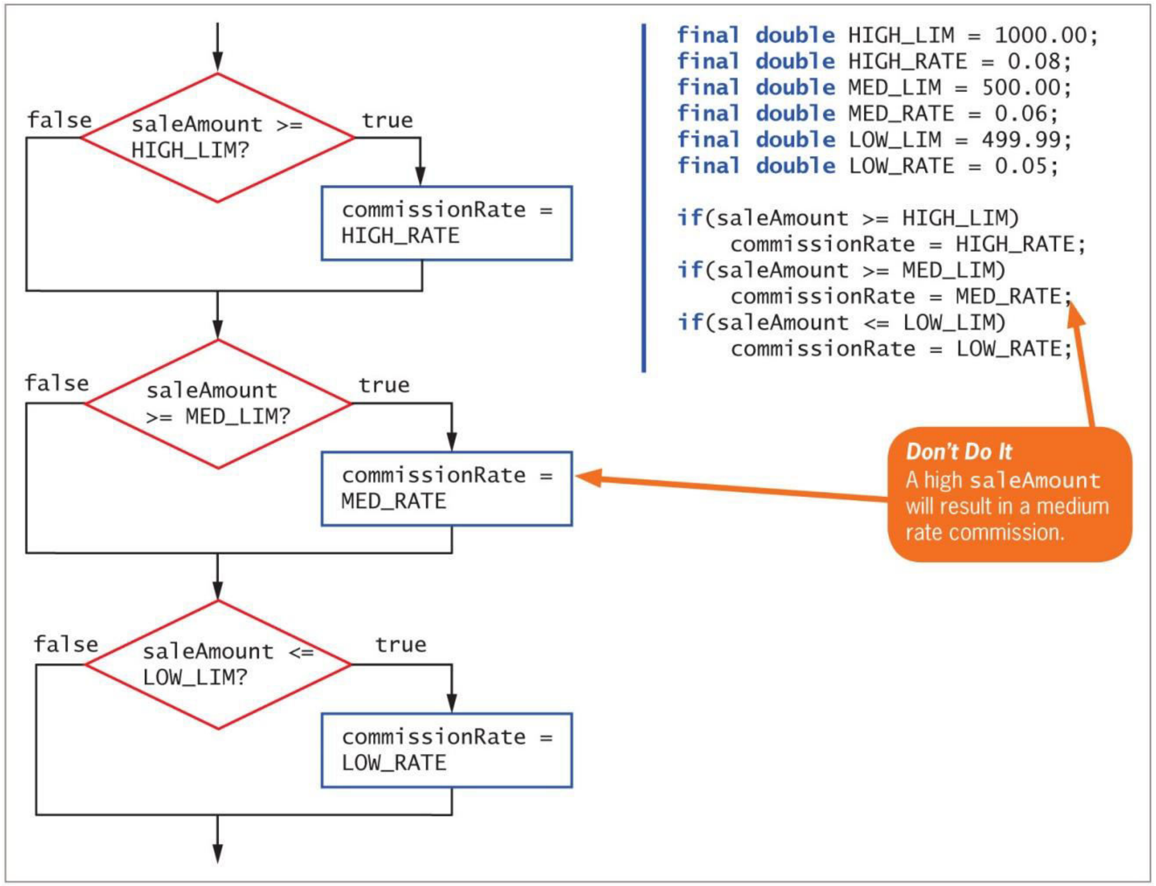
Using Logical AND and OR Operators (5 of 5)

- **Short-circuit evaluation**
  - Expressions on each side of the logical operator are evaluated only as far as necessary
  - Determine whether an expression is `true` or `false`

Making Accurate and Efficient Decisions (1 of 4)

- Making accurate range checks
  - **Range check**: a series of `if` statements that determine whether a value falls within a specified range
  - Java programmers commonly place each `else` of a subsequent `if` on the same line
  - Within a nested `if…else` statement:
    - It is most efficient to ask the most likely question first
    - Avoid asking multiple questions

Making Accurate and Efficient Decisions (2 of 4)


false
saleAmount >= HIGH_LIM?
true
commissionRate = HIGH_RATE
false
saleAmount >= MED_LIM?
true
commissionRate = MED_RATE
false
saleAmount <= LOW_LIM?
true
commissionRate = LOW_RATE
Don't Do It
A high saleAmount will result in a medium rate commission.

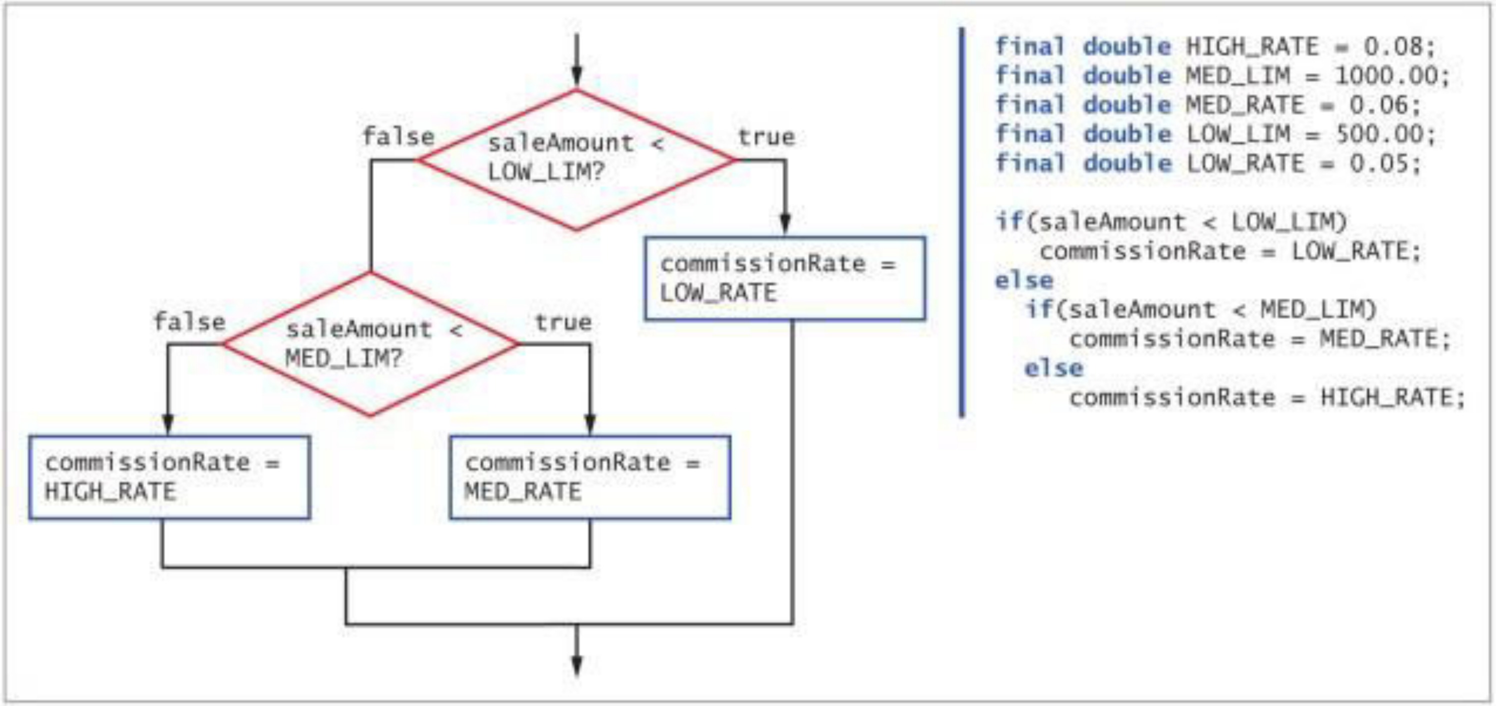
```
final double HIGH_LIM = 1000.00;
final double HIGH_RATE = 0.08;
final double MED_LIM = 500.00;
final double MED_RATE = 0.06;
final double LOW_LIM = 499.99;
final double LOW_RATE = 0.05;

if(saleAmount >= HIGH_LIM)
    commissionRate = HIGH_RATE;
if(saleAmount >= MED_LIM)
    commissionRate = MED_RATE;
if(saleAmount <= LOW_LIM)
    commissionRate = LOW_RATE;
```

Figure 5-19 Incorrect commission-determining code and its logic

Making Accurate and Efficient Decisions (3 of 4)

- It is most efficient to ask a question most likely to be `true` first
  - Avoids asking multiple questions
  - Makes a sequence of decisions more efficient

Making Accurate and Efficient Decisions (4 of 4)


false
saleAmount < LOW_LIM?
true
commissionRate = LOW_RATE
false
saleAmount < MED_LIM?
true
commissionRate = HIGH_RATE
commissionRate = MED_RATE

```
final double HIGH_RATE = 0.08;
final double MED_LIM = 1000.00;
final double MED_RATE = 0.06;
final double LOW_LIM = 500.00;
final double LOW_RATE = 0.05;

if(saleAmount < LOW_LIM)
   commissionRate = LOW_RATE;
else
  if(saleAmount < MED_LIM)
     commissionRate = MED_RATE;
  else
     commissionRate = HIGH_RATE;
```

Figure 5-22 Commission-determining code and logic that evaluates smallest first

Using && and || Appropriately

- Errors of beginning programmers:
  - Using the AND operator when they mean to use OR
    - Example: No `payRate` value can ever be both less than 5.65 and more than 60 at the same time

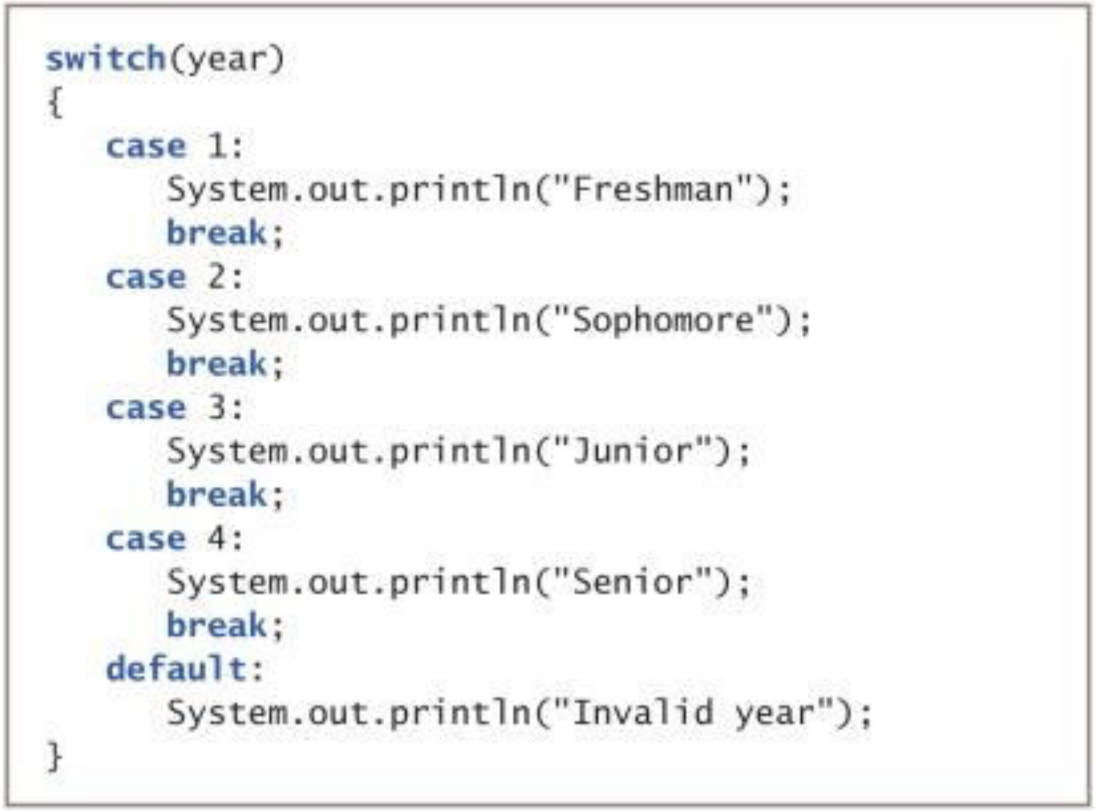
```
if(payRate < LOW && payRate > HIGH)
 System.out.println("Error in pay rate");
```

    - Use pipes “||” operator instead
  - Using a single ampersand or pipe to indicate a logical AND or OR

Using the `switch` Statement (1 of 3)

- `switch` statement
  - An alternative to a series of nested `if` statements
  - Test a single variable against a series of exact integer, character, or string values
- Keywords
  - `switch`
    - Starts the structure
    - Followed by a test expression enclosed in parentheses
  - `case`
    - Followed by one of the possible values for the test expression and a colon
  - `break`
    - Optionally terminates a `switch` statement at the end of each case
  - `default`
    - Optionally is used prior to any action that should occur if the test variable does not match any case

Using the `switch` Statement (2 of 3)



```
switch(year)
{
    case 1:
        System.out.println("Freshman");
        break;
    case 2:
        System.out.println("Sophomore");
        break;
    case 3:
        System.out.println("Junior");
        break;
    case 4:
        System.out.println("Senior");
        break;
    default:
        System.out.println("Invalid year");
}
```

Figure 5-24 Determining class status using a `switch` statement

Using the `switch` Statement (3 of 3)

- `break` statements in the `switch` structure
  - If a `break` statement is omitted:
    - The program finds a match for the test variable
    - All statements within the `switch` statement execute from that point forward
- `case` statement
  - No need to write code for each case
  - Evaluate `char` variables
    - Ignore whether it is uppercase or lowercase
- Why use `switch` statements?
  - They are convenient when several alternative courses of action depend on a single integer, character, or string value
  - Use only when there is a reasonable number of specific matching values to be tested

Using the Conditional and NOT Operators (1 of 2)

- **Conditional operator**
  - Requires three expressions separated with a question mark and a colon
  - Used as an abbreviated version of the `if…else` structure
  - You are never required to use it
- Syntax of a conditional operator:

```
testExpression ? trueResult : falseResult;
```

Using the Conditional and NOT Operators (2 of 2)

- A Boolean expression is evaluated as `true` or `false`
  - If the value of `testExpression` is `true`:
    - The entire conditional expression takes on the value of the expression following the question mark
  - If the value is `false`:
    - The entire expression takes on the value of `falseResult`
- An advantage of using the conditional operator is the conciseness of the statement

Using the NOT Operator

- **NOT operator**
  - Written as an exclamation point (`!`)
  - Negates the result of any Boolean expression
  - When preceded by the NOT operator, any expression evaluated as:
    - `true` **becomes** `false`
    - `false` **becomes** `true`
- Statements with the NOT operator:
  - Are harder to read
  - Require a double set of parentheses

Understanding Operator Precedence (1 of 4)

- Combine as many AND or OR operators as needed
- An operator's precedence
  - How an expression is evaluated
  - The order agrees with common algebraic usage
    - Arithmetic is done first
    - Assignment is done last
    - The AND operator is evaluated before the OR operator
    - Statements in parentheses are evaluated first

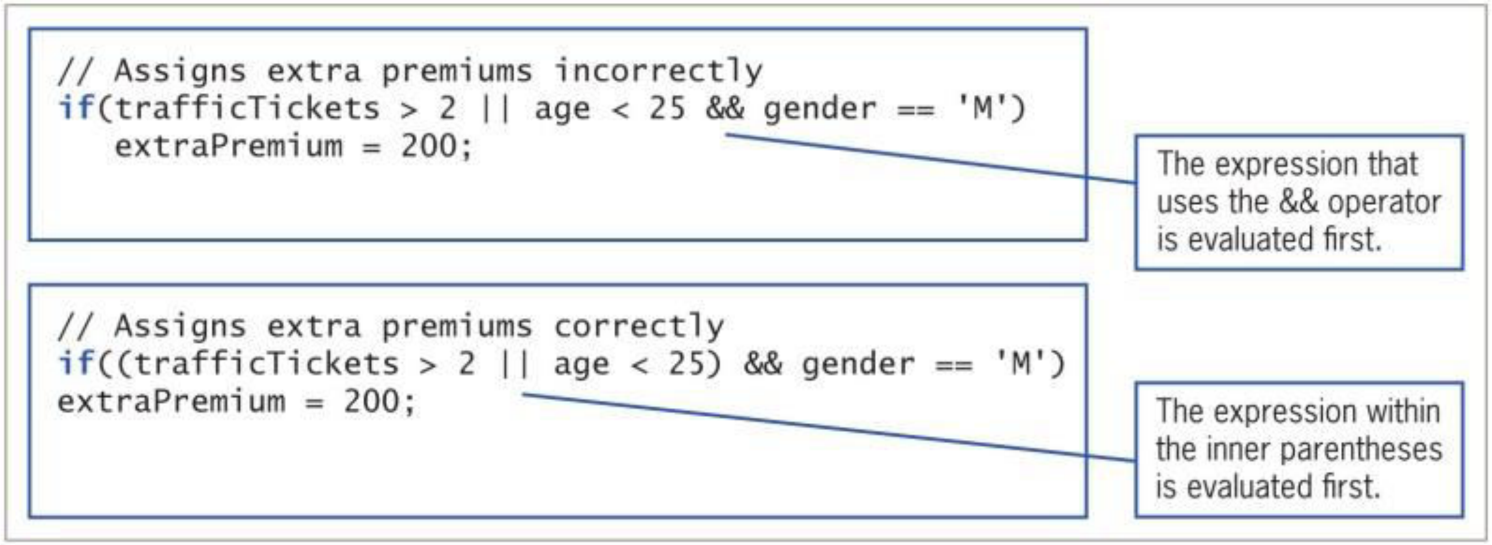
Understanding Operator Precedence (2 of 4)

Table 5-1 Operator precedence for operators used so far		
Precedence	**Operator(s)**	**Symbol(s)**
Highest	Logical NOT	!
Intermediate	Multiplication, division, modulus Addition, subtraction Relational Equality Logical AND Logical OR Conditional	*/% + – > < >= <= == != && || ?:
Lowest	Assignment	=

Understanding Operator Precedence (3 of 4)

- Two important conventions
  - The order in which operators are used makes a difference
  - Always use parentheses to change precedence or make your intentions clearer

Understanding Operator Precedence (4 of 4)



```
// Assigns extra premiums incorrectly
if(trafficTickets > 2 || age < 25 && gender == 'M')
   extraPremium = 200;
```

The expression that uses the && operator is evaluated first.

```
// Assigns extra premiums correctly
if((trafficTickets > 2 || age < 25) && gender == 'M')
extraPremium = 200;
```

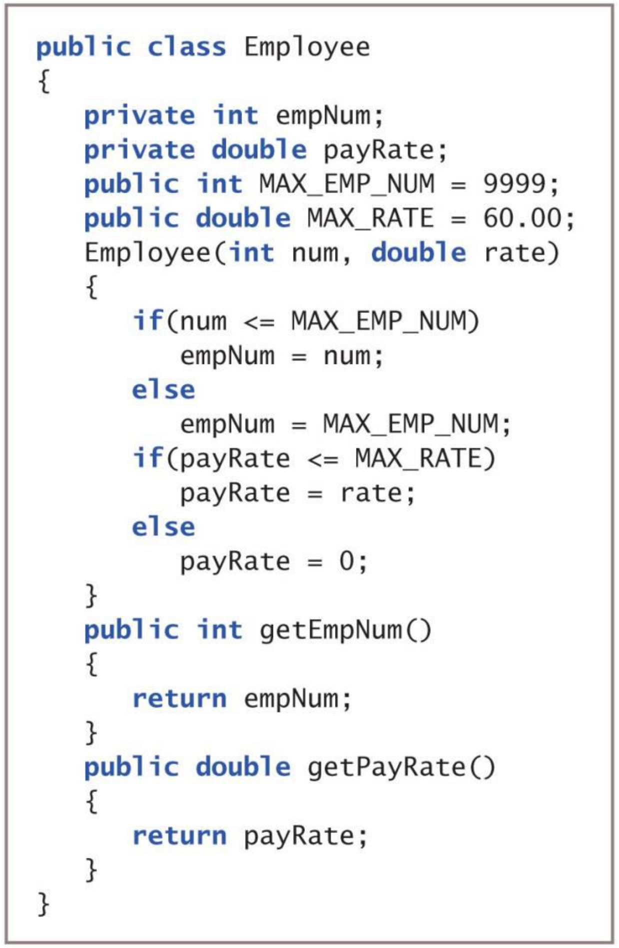
The expression within the inner parentheses is evaluated first.

Figure 5-31 Two comparisons using && and ||

Adding Decisions and Constructors to Instance Methods (1 of 2)

- Helps ensure that fields have acceptable values
- Determines whether values are within the allowed limits for the fields

Adding Decisions and Constructors to Instance Methods (2 of 2)



```java
public class Employee
{
   private int empNum;
   private double payRate;
   public int MAX_EMP_NUM = 9999;
   public double MAX_RATE = 60.00;
   Employee(int num, double rate)
   {
      if(num <= MAX_EMP_NUM)
         empNum = num;
      else
         empNum = MAX_EMP_NUM;
      if(payRate <= MAX_RATE)
         payRate = rate;
      else
         payRate = 0;
   }
   public int getEmpNum()
   {
      return empNum;
   }
   public double getPayRate()
   {
      return payRate;
   }
}
```

Figure 5-32 The `Employee` class that contains a constructor that makes decisions

Don’t Do It (1 of 2)

- Don’t ignore subtleties in boundaries used in decision making
- Don’t use the assignment operator instead of the comparison operator
- Don’t insert a semicolon after the Boolean expression in an `if` statement
- Don’t forget to block a set of statements with curly braces when several statements depend on the `if` or the `else` statement

Don’t Do It (2 of 2)

- Don’t forget to include a complete Boolean expression on each side of an `&&` or `||` operator
- Don’t try to use a `switch` structure to test anything other than an integer, a character, or a string value
- Don’t forget a `break` statement if one is required
- Don’t use the standard relational operators to compare objects

Summary (1 of 3)

- `if` statement
  - Makes a decision based on a Boolean expression
- Single-alternative `if`
  - Performs an action based on one alternative
- Dual-alternative `if`
  - `if…else`
  - Performs one action when a Boolean expression evaluates as `true`
  - Performs a different action when an expression evaluates as `false`

Summary (2 of 3)

- AND operator
  - `&&`
  - Determines whether two expressions are both `true`
- OR operator
  - `||`
  - Carries out some action even if only one of two conditions is `true`
- `switch` statement
  - Tests a single variable against a series of exact integer or character values

CENGAGE

Summary (3 of 3)

- Conditional operator
  - An abbreviated version of an `if…else` statement
- NOT operator
  - `!`
  - Negates the result of any Boolean expression
- Operator precedence