JAVA™ PROGRAMMING

Chapter 5: Making Decisions

Objectives

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

Objectives (cont'd.)

- 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

- **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 (cont'd.)

- **Flowchart** (cont'd.)
  - Programmers use a variety of shapes to represent 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

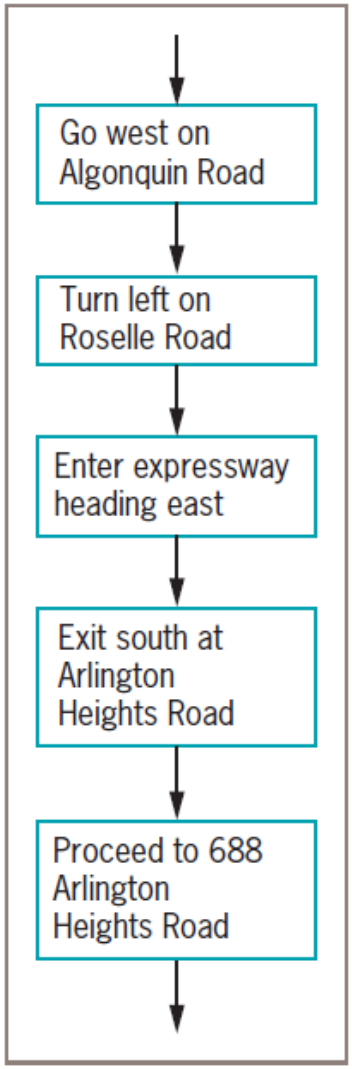


Figure 5-1 Flowchart of a series of sequential steps

Planning Decision-Making Logic (cont'd.)

- **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

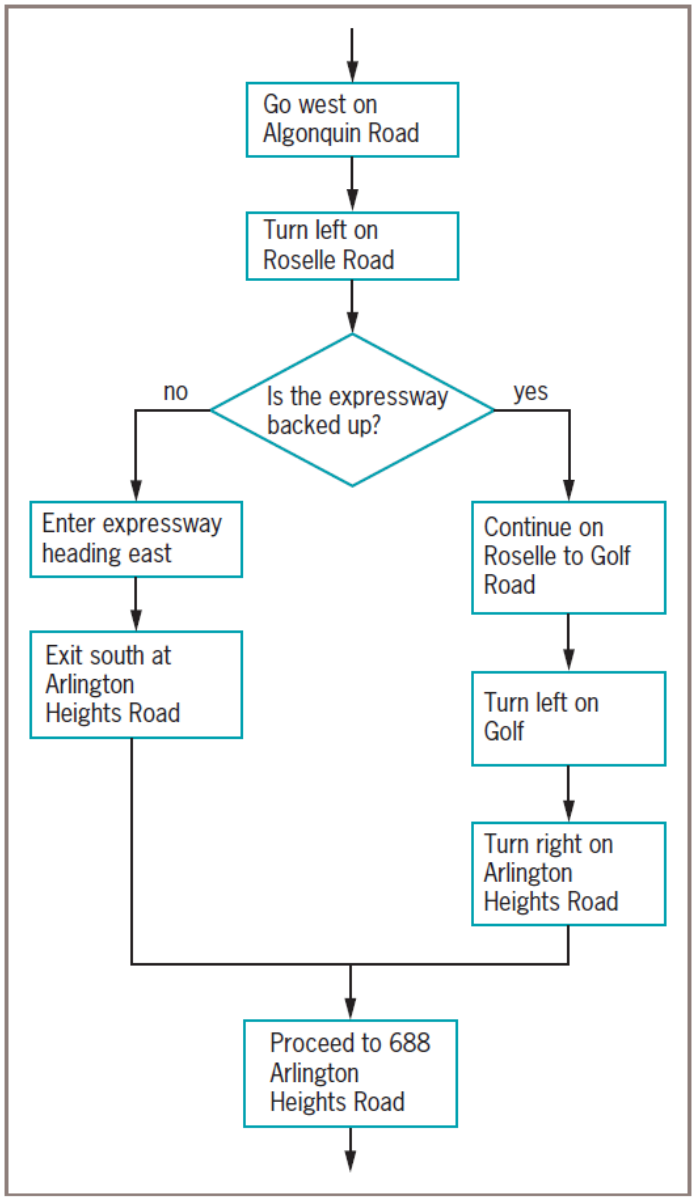


Figure 5-2 Flowchart including a decision

The `if` and `if…else` Structures

- **`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` Structures (cont'd.)

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

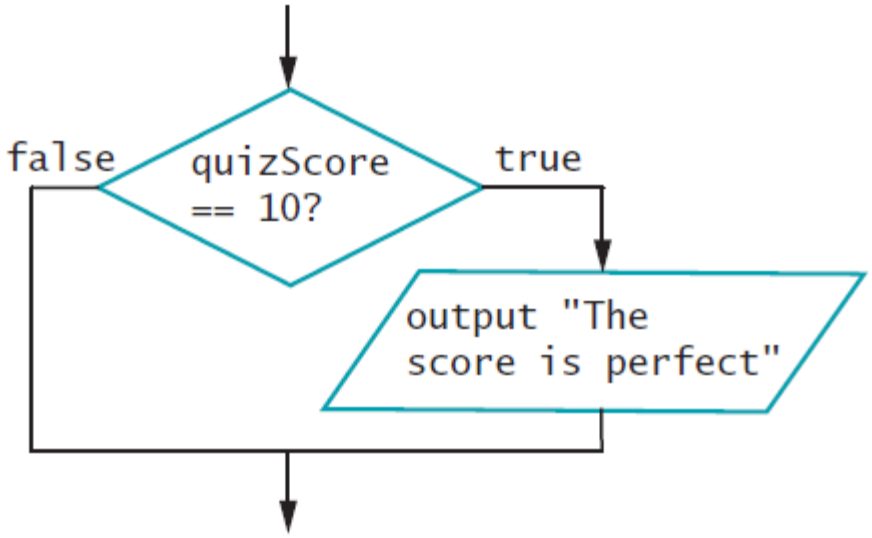


Figure 5-3 A Java `if` statement

Pitfall: Misplacing a Semicolon in an `if` Statement

- 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 (cont'd.)

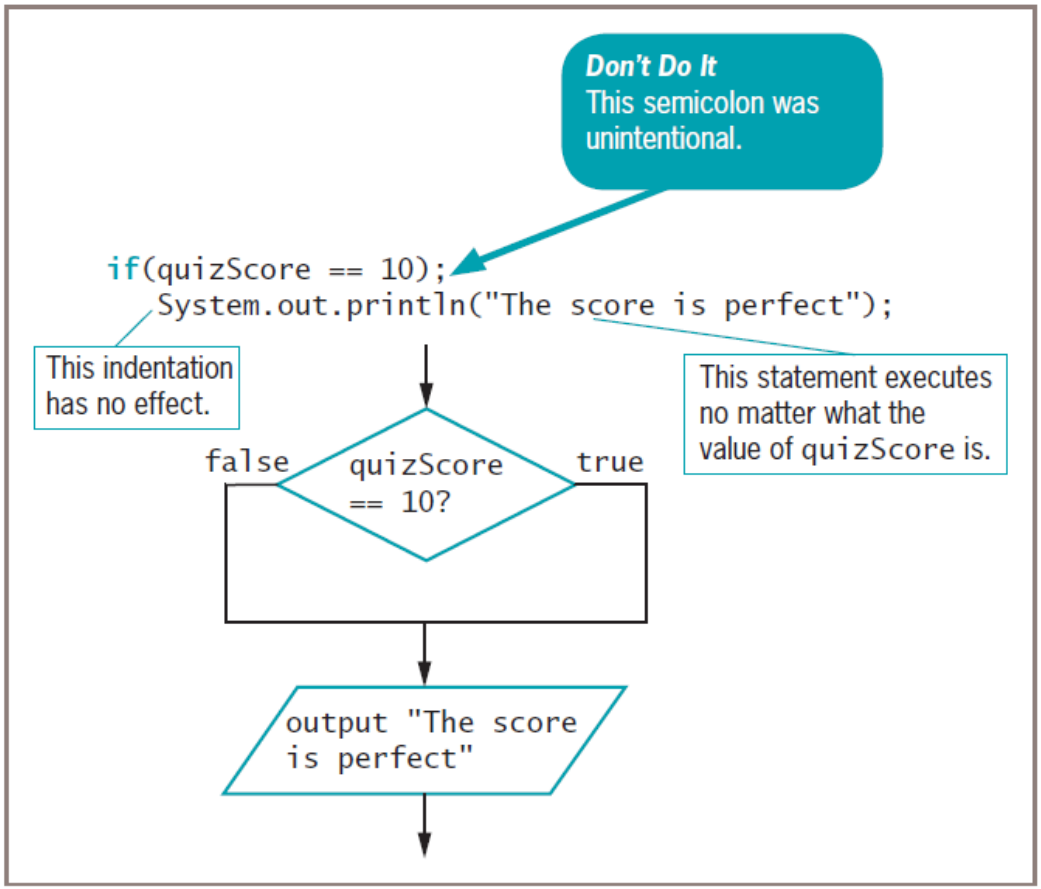


Figure 5-4 Logic that executes when an extra semicolon is inserted in an `if` statement

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` Structure

- **Single-alternative `if`**
  - Only perform 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` Structure (cont'd.)

- **`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`
  - It's 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` Structure (cont'd.)

```
if(quizScore == 10)
   System.out.println("The score is perfect");
else
   System.out.println("No, it's not");
```

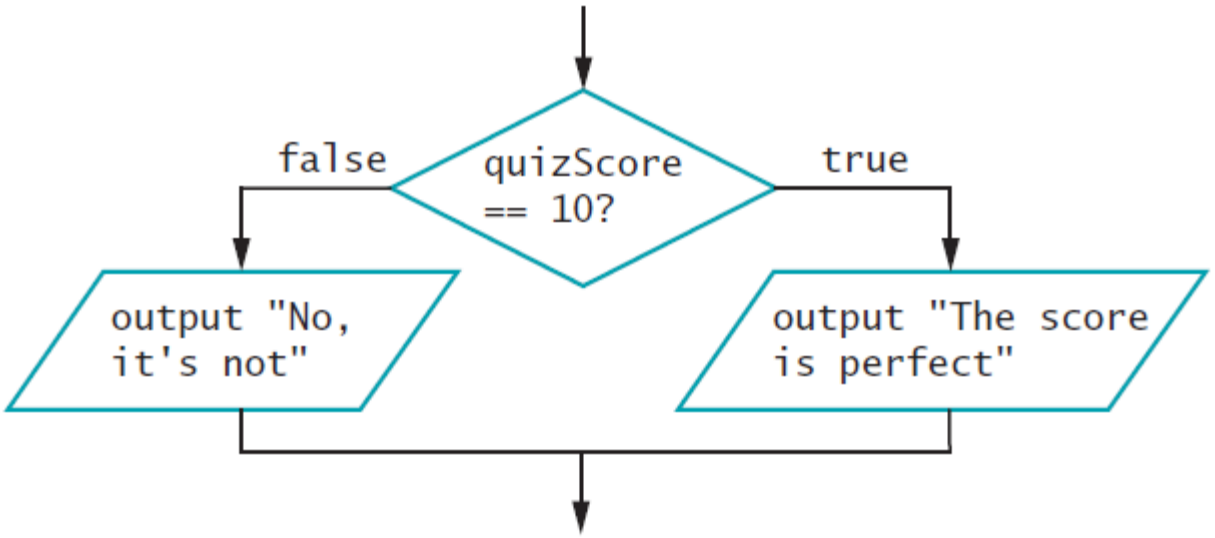


Figure 5-5 An `if...else` structure

Using Multiple Statements in `if` and `if…else` Clauses

- To execute more than one statement, use a pair of curly braces
  - Place dependent statements within a block
  - It's 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 (cont'd.)

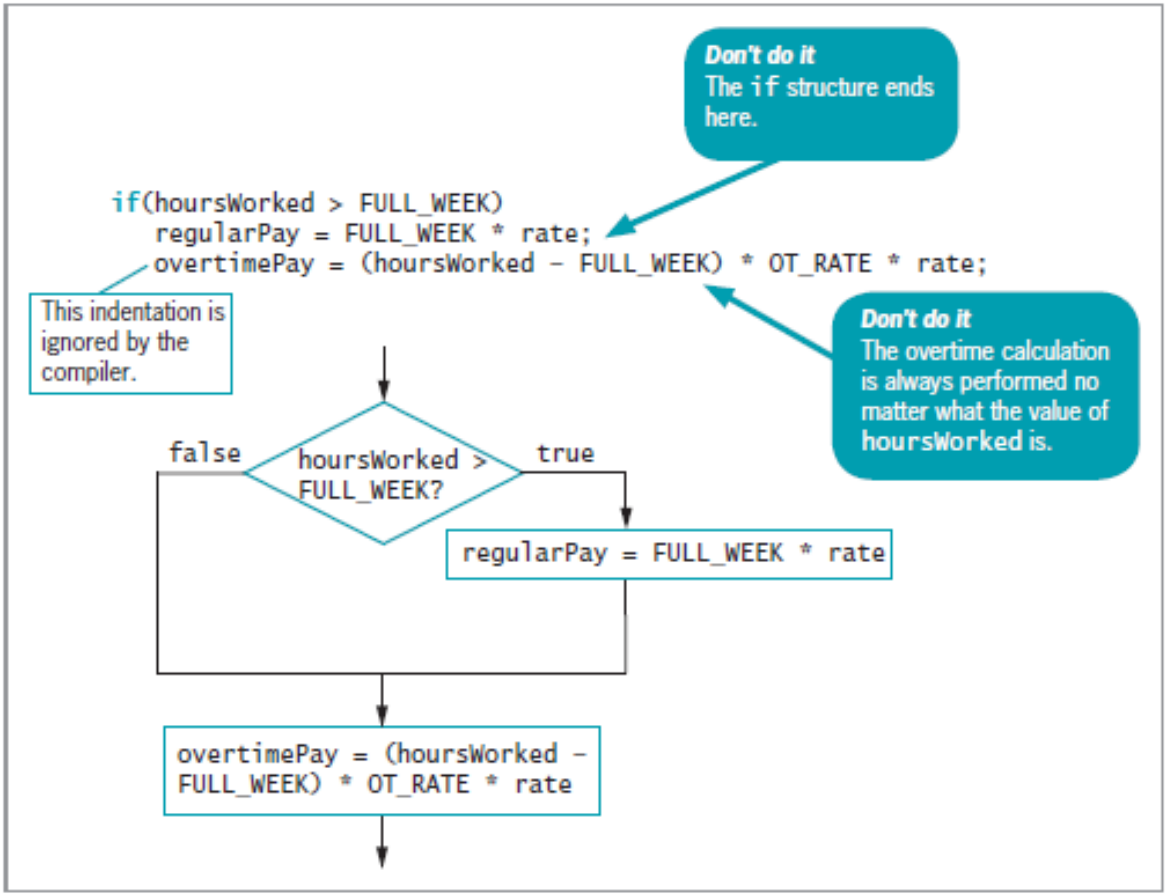


Figure 5-8 Erroneous overtime pay calculation with missing curly braces

Nesting `if` and `if…else` Statements

- **Nested `if` statements**
  - Statements in which an `if` structure is contained inside another `if` structure
  - Use when 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 (cont'd.)

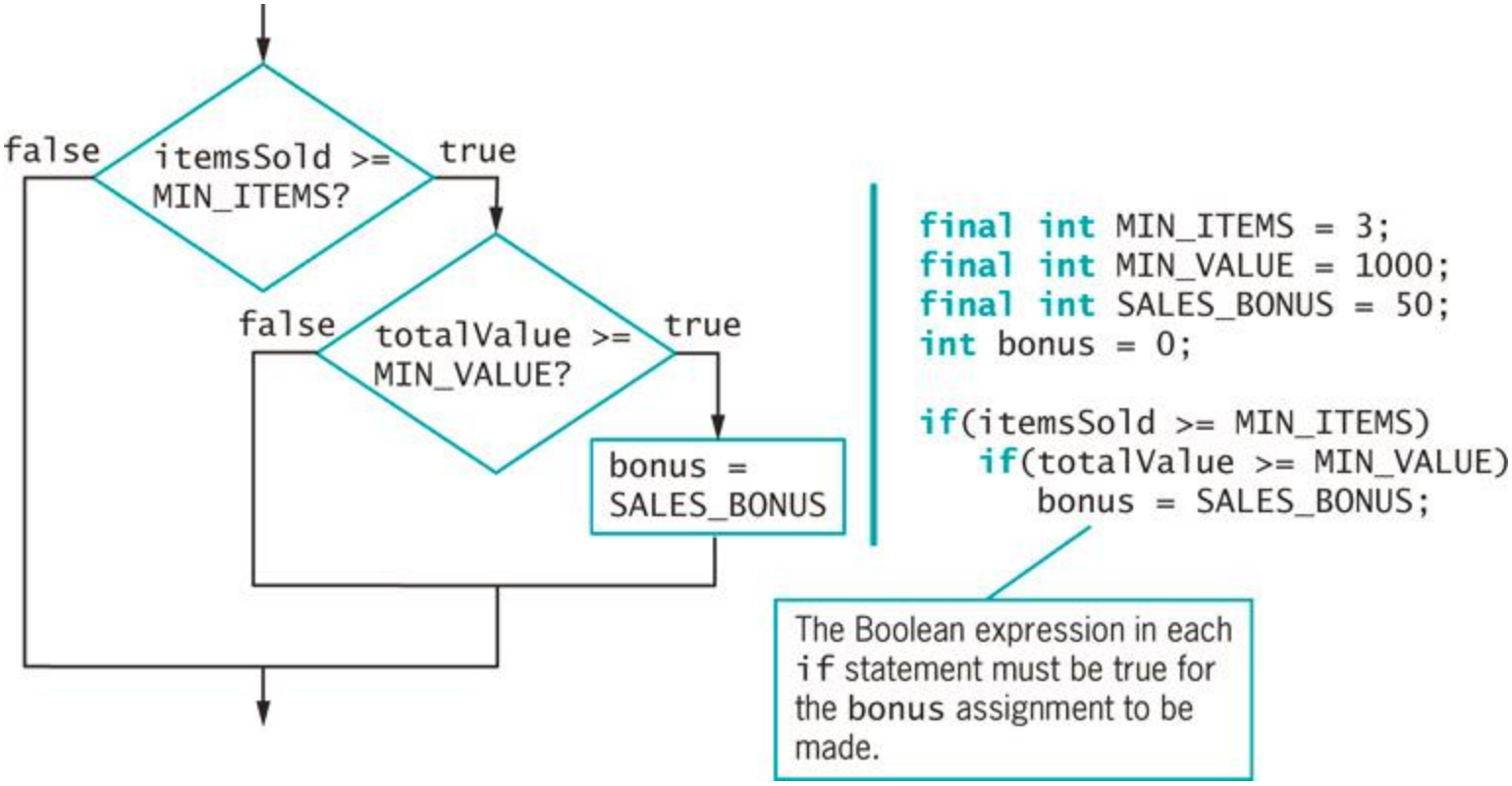
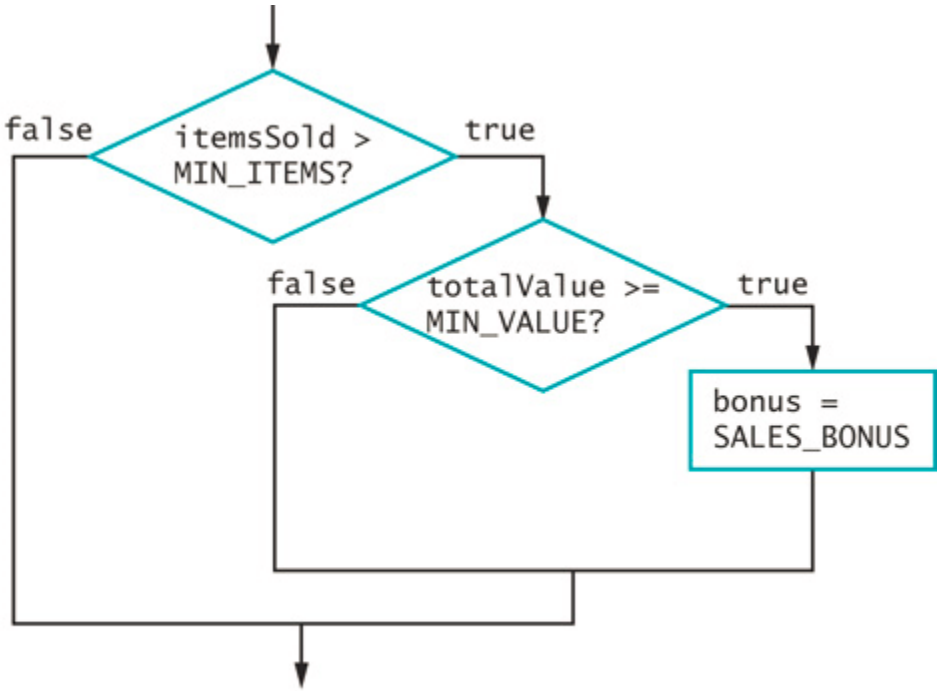


Figure 5-12 Determining whether to assign a bonus using nested `if` statements

Using Logical AND and OR Operators

- 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 (cont'd.)




```
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 for bonus-determining decision using nested `if`s and using the `&&` operator

Using Logical AND and OR Operators (cont'd.)

- 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 (cont'd.)

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

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

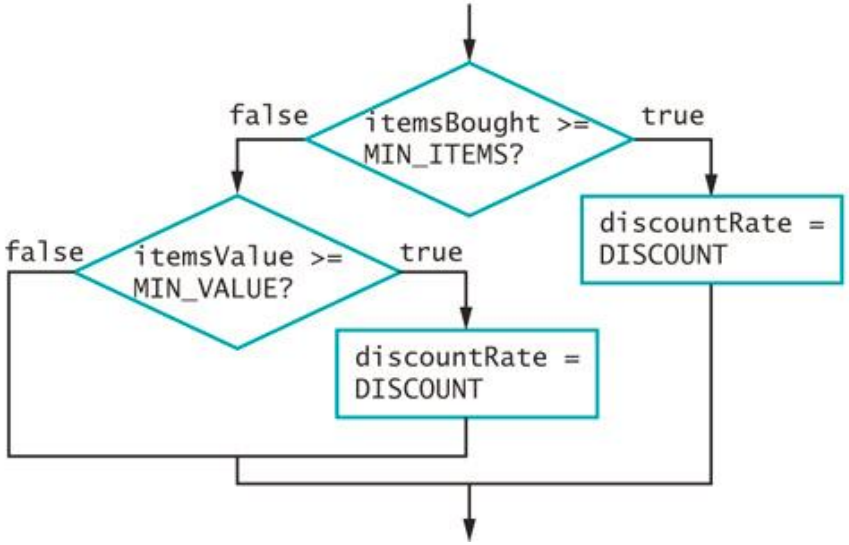


Figure 5-16 Determining customer discount when customer needs to meet only one of two criteria

Using Logical AND and OR Operators (cont'd.)

- **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

- 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's most efficient to ask the most likely question first
    - Avoid asking multiple questions

Making Accurate and Efficient Decisions (cont'd.)

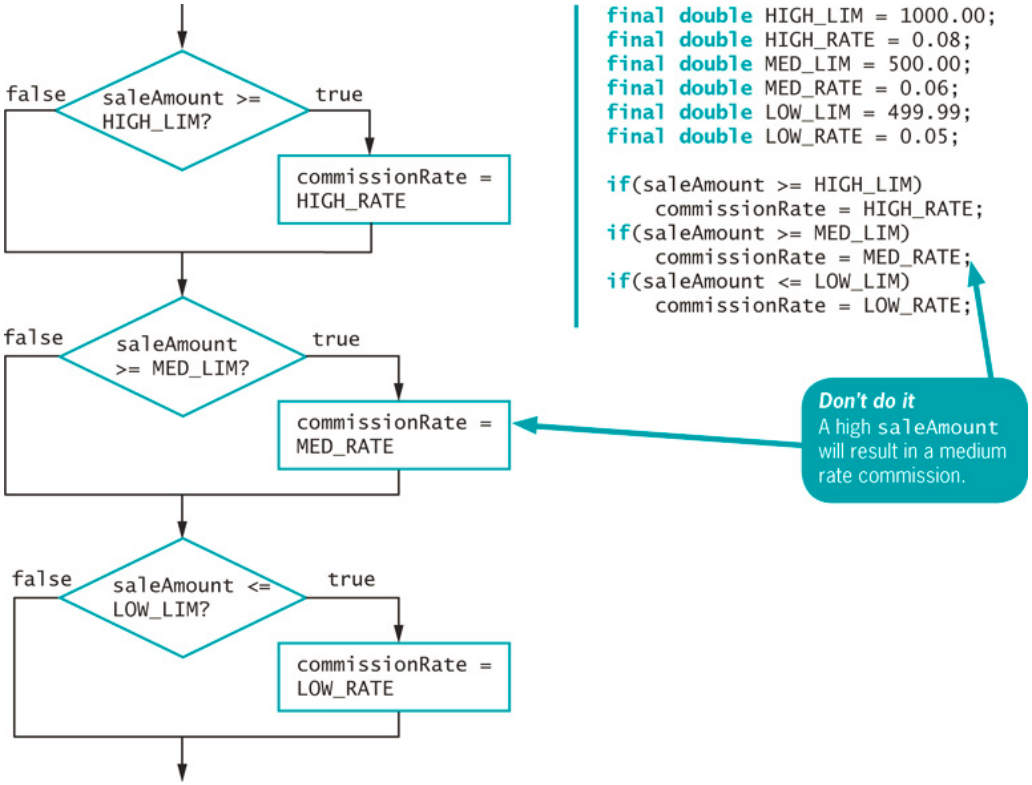
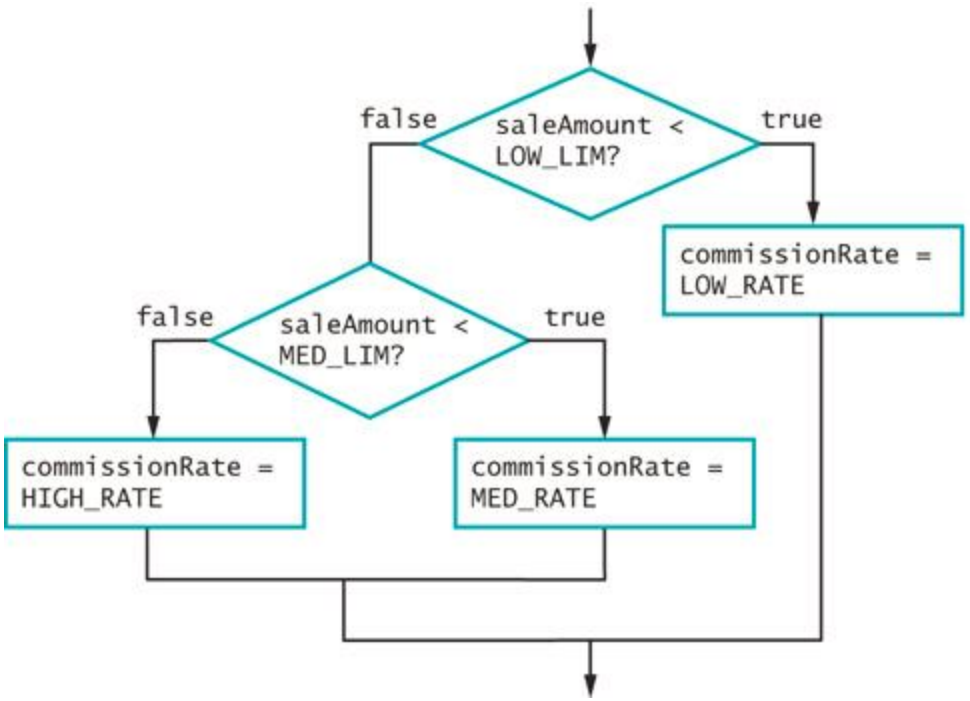


Figure 5-19 Incorrect commission-determining code

Making Accurate and Efficient Decisions (cont'd.)

- It's 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 (cont'd.)




```
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 asking about smallest `saleAmount` 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

- `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

Using the `switch` Statement (cont'd.)

- 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

Using the `switch` Statement (cont'd.)

- Keywords (cont'd.)
  - `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 (cont'd.)

```
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 (cont'd.)

- `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's uppercase or lowercase

Using the `switch` Statement (cont'd.)

- 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

- **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 (cont'd.)

- 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

- 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 (cont'd.)

Precedence	Operator(s)	Symbol(s)
Highest	Logical NOT	!
Intermediate	Multiplication, division, modulus	* / %
	Addition, subtraction	+ −
	Relational	> < >= <=
	Equality	== !=
	Logical AND	&&
	Logical OR	||
	Conditional	?:
Lowest	Assignment	=

Table 5-1 Operator precedence for operators used so far

Understanding Operator Precedence (cont'd.)

- 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 (cont'd.)

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

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 5-30 Two comparisons using && and ||

Adding Decisions and Constructors to Instance Methods

- 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 (cont'd.)

```
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 5-31 The `Employee` class that contains a constructor that makes decisions

You Do It

- Using an `if…else` Statement
- Using Multiple Statements in `if` and `else` Clauses
- Using a Nested `if` Statement
- Using the && Operator
- Using the `switch` Statement
- Adding Decisions to Constructors and Instance Methods

Don’t Do It

- 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 (cont’d.)

- 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

- `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 (cont'd.)

- 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

Summary (cont'd.)

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

JAVA™ PROGRAMMING

Chapter 6: Looping

Objectives

- Learn about the loop structure
- Create `while` loops
- Use shortcut arithmetic operators
- Create `for` loops
- Create `do…while` loops
- Nest loops
- Improve loop performance

Learning About the Loop Structure

- **Loop**
  - A structure that allows repeated execution of a block of statements
- **Loop body**
  - A block of statements
  - Executed repeatedly
- **Iteration**
  - One execution of any loop

Learning About the Loop Structure (cont'd.)

- Three types of loops
  - `while`
    - The loop-controlling Boolean expression is the first statement
  - `for`
    - A concise format in which to execute loops
  - `do…while`
    - The loop-controlling Boolean expression is the last statement

Learning About the Loop Structure (cont'd.)

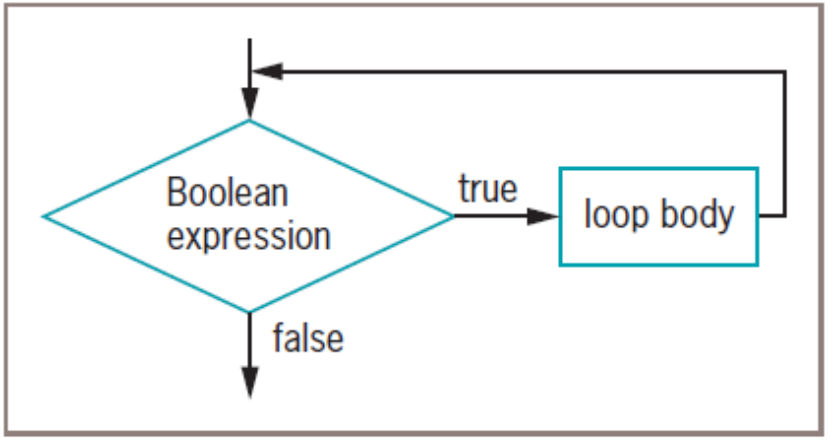


Figure 6-1 Flowchart of a loop structure

Creating `while` Loops

- **`while` loop**
  - Executes a body of statements continually
    - As long as the Boolean expression that controls entry into the loop continues to be `true`
  - Consists of:
    - The keyword `while`
    - Followed by a Boolean expression within parentheses
    - Followed by the body of the loop; can be a single statement or a block of statements surrounded by curly braces

Writing a Definite `while` Loop

- **Definite loop**
  - Performs a task a predetermined number of times
  - Also called a counted loop
- Write a definite loop
  - Initialize the loop control variable
    - The variable whose value determines whether loop execution continues
  - While the loop control variable does not pass a limiting value, the program continues to execute the body of the `while` loop

Writing a Definite `while` Loop (cont'd.)

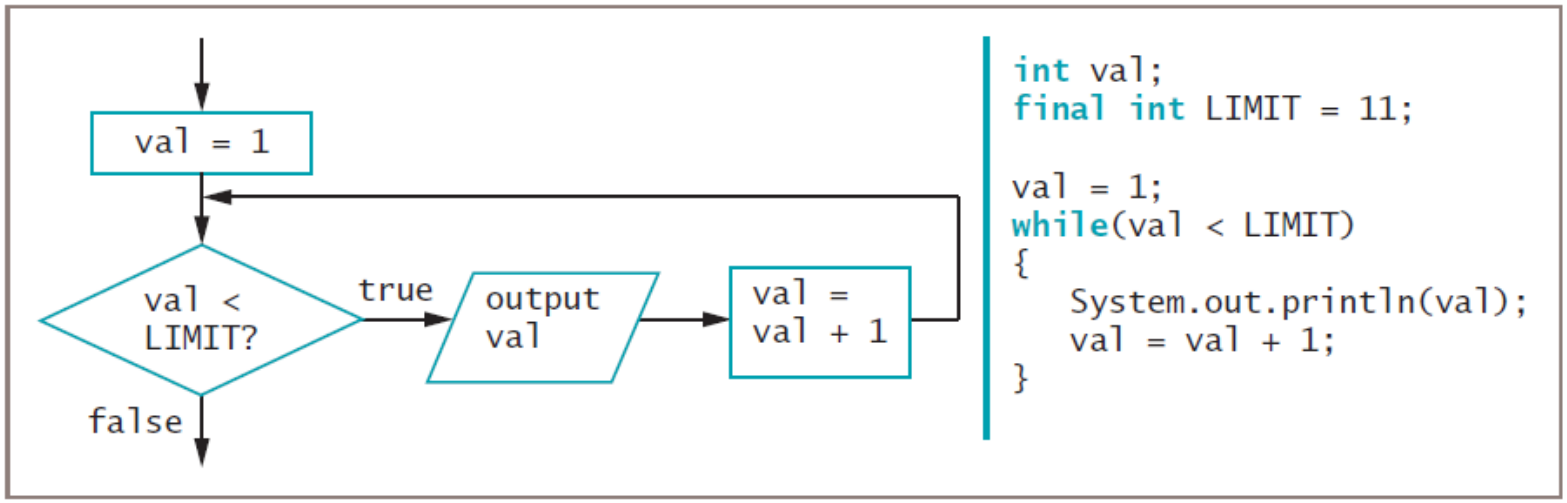


Figure 6-2 A `while` loop that displays the integers 1 through 10

Writing a Definite `while` Loop (cont'd.)

- Write a definite loop (cont'd.)
  - The body of the loop must include a statement that alters the loop control variable
- **Infinite loop**
  - A loop that never ends
  - Can result from a mistake in the `while` loop
  - Do not write intentionally

Writing a Definite `while` Loop (cont'd.)

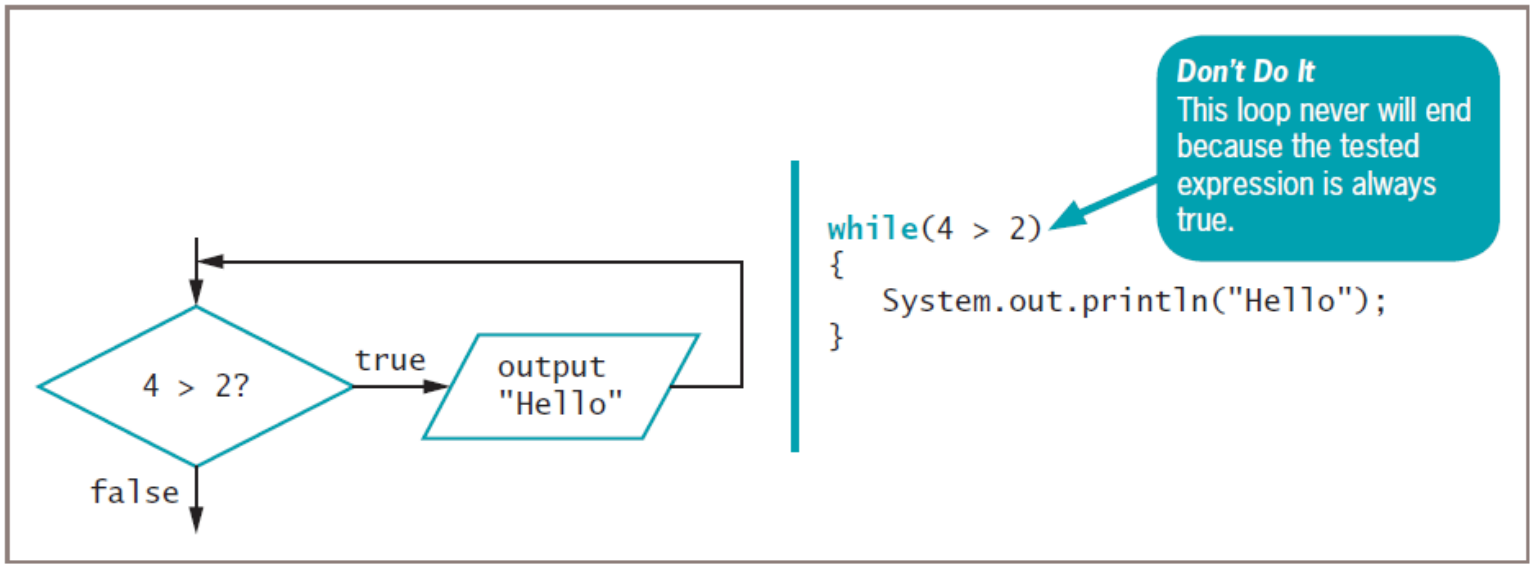


Figure 6-3 A loop that displays "Hello" infinitely

Writing a Definite `while` Loop (cont'd.)

- Suspect an infinite loop when:
  - The same output is displayed repeatedly
  - The screen remains idle for an extended period of time
- To exit an infinite loop, press and hold Ctrl, then press C or Break

Writing a Definite `while` Loop (cont'd.)

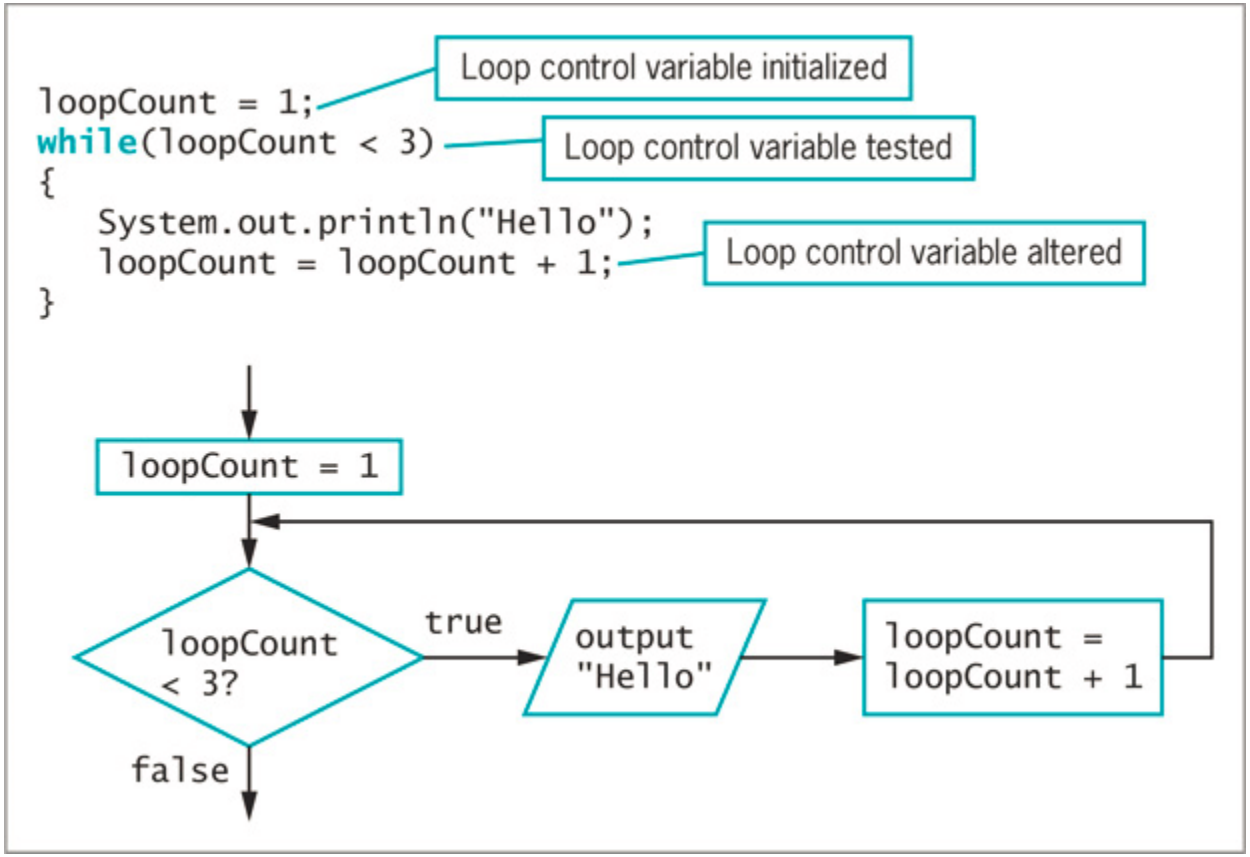


Figure 6-4 A `while` loop that displays "Hello" twice

Pitfall: Failing to Alter the Loop Control Variable Within the Loop Body

- Prevent the `while` loop from executing infinitely
  - The named loop control variable is initialized to a starting value
  - The loop control variable is tested in the `while` statement
  - If the test expression is `true`, the body of the `while` statement takes action
    - Alters the value of the loop control variable
  - The test of the `while` statement must eventually evaluate to `false`

Pitfall: Failing to Alter the Loop Control Variable Within the Loop Body (cont'd.)

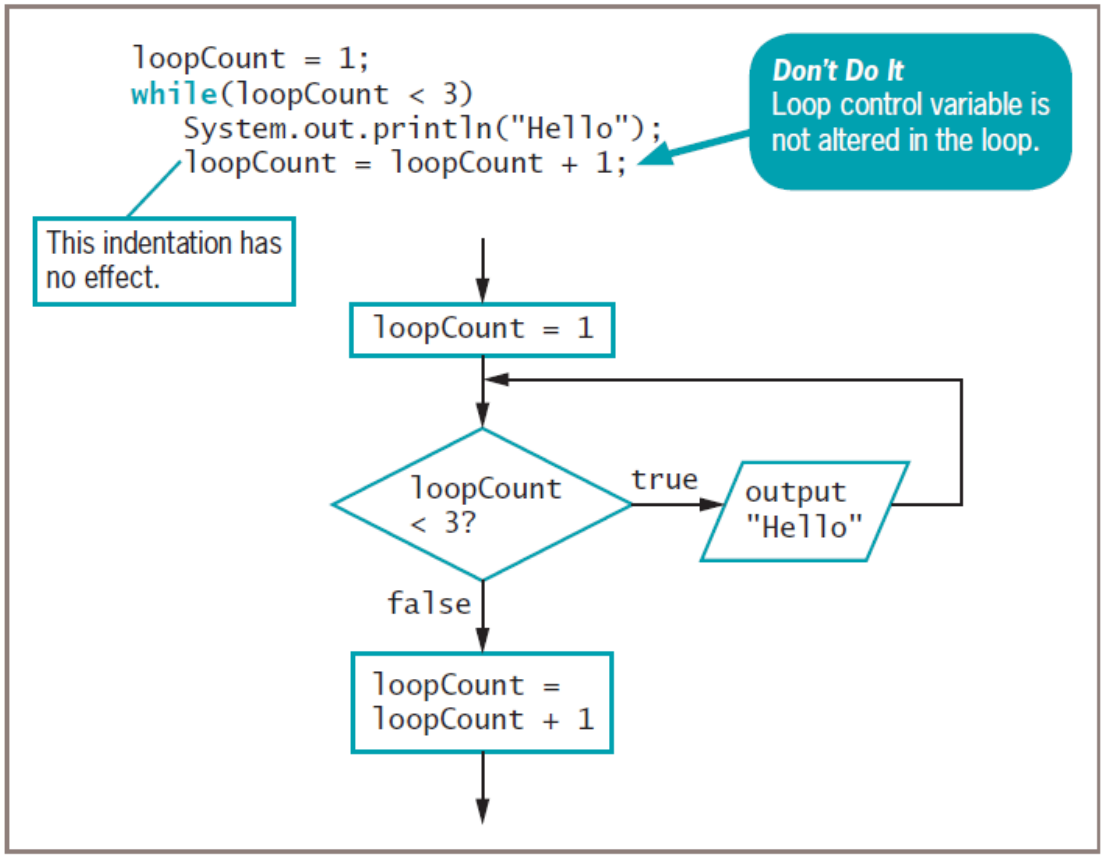


Figure 6-5 A `while` loop that displays "Hello" infinitely because `loopCount` is not altered in the loop body

Pitfall: Creating a Loop with an Empty Body

- Loop control variable
  - A variable that is altered and stored with a new value

```
loopCount = loopCount + 1
```

    - The equal sign assigns a value to the variable on the left
  - The variable should be altered within the body of the loop
- **Empty body**
  - A body with no statements
  - Caused by misplaced semicolons

Pitfall: Creating a Loop with an Empty Body (cont'd.)

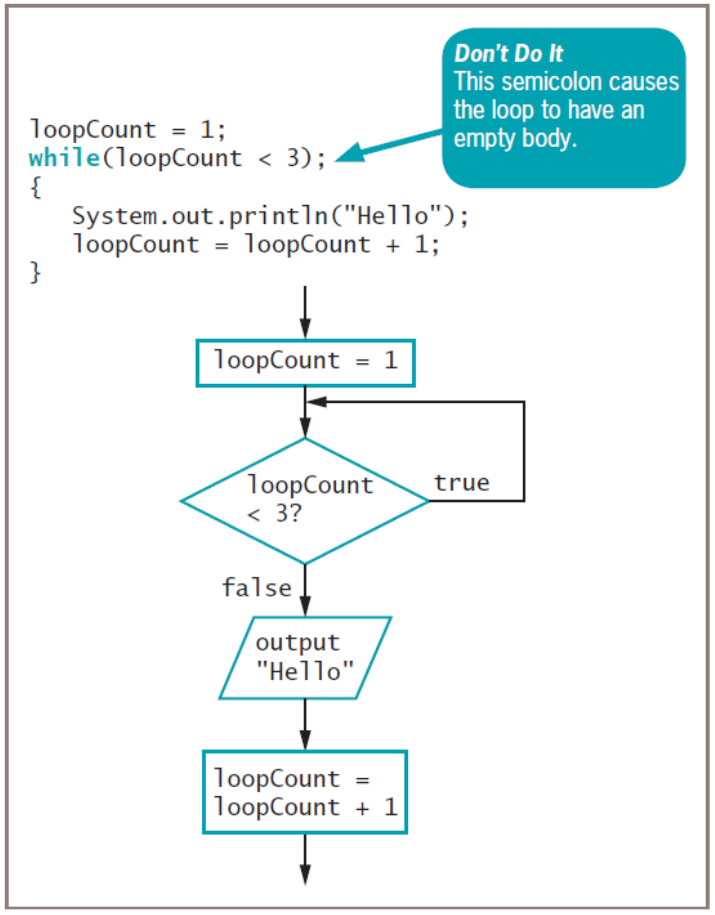


Figure 6-6 A `while` loop that loops infinitely with no output because the loop body is empty

Altering a Definite Loop's Control Variable

- **Incrementing** the variable
  - Alter the value of the loop control variable by adding 1
- **Decrementing** the variable
  - Subtract 1 from the loop control variable
- Clearest and best method
  - Start the loop control variable at 0 or 1
  - Increment by 1 each time through the loop
  - Stop when the loop control variable reaches the limit

Altering a Definite Loop's Control Variable (cont'd.)

```
loopCount = 3;
while(loopCount > 1)
{
   System.out.println("Hello");
   loopCount = loopCount - 1;
}
```

Figure 6-7 A `while` loop that displays "Hello" twice, decrementing the `loopCount` variable in the loop body

Writing an Indefinite `while` Loop

- Indefinite loop
  - Altered by user input
    - Controlled by the user
    - Executed any number of times
- Validating data
  - Ensure a value falls within a specified range
  - Use indefinite loops to validate input data
  - If a user enters incorrect data, the loop repeats

Writing an Indefinite `while` Loop (cont'd.)

```
import java.util.Scanner;
public class BankBalance
{
   public static void main(String[] args)
   {
      double balance;
      int response;
      int year = 1;
      final double INT_RATE = 0.03;
      Scanner keyboard = new Scanner(System.in);
      System.out.print("Enter initial bank balance > ");
      balance = keyboard.nextDouble();
      System.out.println("Do you want to see next year's balance?");
      System.out.print("Enter 1 for yes");
      System.out.print("   or any other number for no >> ");
      response = keyboard.nextInt();
      while(response == 1)
      {
         balance = balance + balance * INT_RATE;
         System.out.println("After year " + year + " at " +  INT_RATE +
             " interest rate, balance is $" + balance);
         year = year + 1;
         System.out.println("\nDo you want to see the balance " +
            "at the end of another year?");
         System.out.print("Enter 1 for yes");
         System.out.print("   or any other number for no >> ");
         response = keyboard.nextInt();
      }
   }
}
```

Figure 6-8 The `BankBalance` application

Validating Data

- Ensuring data falls within a specific range
- **Priming read**
  - Input retrieved before the loop is entered
  - Within a loop, the last statement retrieves the next input value and checks the value before the next entrance of the loop

Validating Data (cont'd.)

```
import java.util.Scanner;
public class EnterSmallValue
{
   public static void main(String[] args)
   {
      int userEntry;
      final int LIMIT = 3;
      Scanner input = new Scanner(System.in);
      System.out.print("Please enter an integer no higher than " +
         LIMIT + " > ");
      userEntry = input.nextInt();
      while(userEntry > LIMIT)
      {
         System.out.println("The number you entered was too high");
         System.out.print("Please enter an integer no higher than " +
            LIMIT + " > ");
         userEntry = input.nextInt();
      }
      System.out.println("You correctly entered " + userEntry);
   }
}
```

Figure 6-10 The `EnterSmallValue` application

Using Shortcut Arithmetic Operators

- **Accumulating**
  - Repeatedly increasing a value by some amount
- Java provides shortcuts for incrementing and accumulating

  `+=` **add and assign operator**

  `-=` **subtract and assign operator**

  `*=` **multiply and assign operator**

  `/=` **divide and assign operator**

  `%=` **remainder and assign operator**

Using Shortcut Arithmetic Operators (cont'd.)

- **Prefix increment operator** and **postfix increment operator**

  `++someValue`, `someValue++`

  - Use only with variables
  - Unary operators
    - Use with one value
  - Increase a variable's value by 1
    - No difference between operators (unless other operations are in the same expression)

Using Shortcut Arithmetic Operators (cont'd.)

```
int value;
value = 24;
++value;  // Result: value is 25
value = 24;
value++;  // Result: value is 25
value = 24;
value = value + 1;   // Result: value is 25
value = 24;
value += 1;  // Result: value is 25
```

Figure 6-13 Four ways to add 1 to a value

Using Shortcut Arithmetic Operators (cont'd.)

- **Prefix increment operator** and **postfix increment operator** (cont'd.)
  - **Prefix ++**
    - The result is calculated and stored
    - Then the variable is used
  - **Postfix ++**
    - The variable is used
    - Then the result is calculated and stored
- Prefix and postfix decrement operators

```
--someValue
someValue--
```

  - Similar logic to increment operators

Using Shortcut Arithmetic Operators (cont'd.)

```
public class IncrementDemo
{
   public static void main(String[] args)
   {
      int myNumber, answer;
      myNumber = 17;
      System.out.println("Before incrementing, myNumber is " +
         myNumber);
      answer = ++myNumber;
      System.out.println("After prefix increment, myNumber is " +
         myNumber);
      System.out.println("  and answer is " + answer);
      myNumber = 17;
      System.out.println("Before incrementing, myNumber is " +
         myNumber);
      answer = myNumber++;
      System.out.println("After postfix increment, myNumber is " +
         myNumber);
      System.out.println("  and answer is " + answer);
   }
}
```

Figure 6-14 The `IncrementDemo` application

Creating a `for` Loop

- **`for` loop**
  - Used when a definite number of loop iterations is required
  - One convenient statement indicates:
    - The starting value for the loop control variable
    - The test condition that controls loop entry
    - The expression that alters the loop control variable

Creating a `for` Loop (cont'd.)

```
for(val = 1; val < 11; ++val)
{
   System.out.println(val);
}

val = 1;
while(val < 11)
{
   System.out.println(val);
   ++val;
}
```

Figure 6-18 A `for` loop and a `while` loop that display the integers 1 through 10

Creating a `for` Loop (cont'd.)

- Other uses for the three sections of a `for` loop
  - Initialization of more than one variable
    - Place commas between separate statements
  - Performance of more than one test using AND or OR operators
  - Decrementation or performance of some other task
  - Altering more than one value
- You can leave one or more portions of a `for` loop empty
    - Two semicolons are still required as placeholders

Creating a `for` Loop (cont'd.)

- Use the same loop control variable in all three parts of a `for` statement
- To pause a program:
  - Use the `for` loop that contains no body

```
for(x = 0; x < 100000; ++x);
```

  - Or use the built-in `sleep()` method

Learning How and When to Use a `do...while` Loop

- **`do...while` loop**
  - A **posttest loop**
  - Checks the value of the loop control variable
    - At the bottom of the loop
    - After one repetition has occurred
  - Performs a task at least one time
  - You are never required to use this type of loop
  - Use curly braces to block the statement
    - Even with a single statement

Learning How and When to Use a `do...while` Loop (cont'd.)

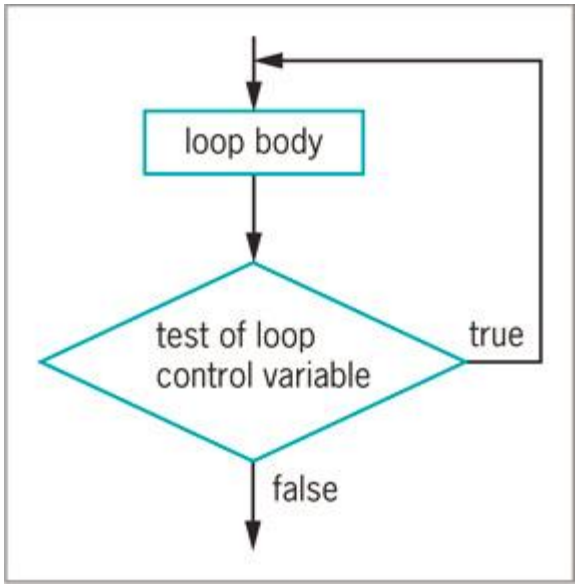


Figure 6-20 General structure of a `do...while` loop

Learning How and When to Use a `do...while` Loop (cont'd.)

```
import java.util.Scanner;
public class BankBalance2
{
   public static void main(String[] args)
   {
      double balance;
      int response;
      int year = 1;
      final double INT_RATE = 0.03;
      Scanner keyboard = new Scanner(System.in);
      System.out.print("Enter initial bank balance > ");
      balance = keyboard.nextDouble();
      keyboard.nextLine();
      do
      {
         balance = balance + balance * INT_RATE;
         System.out.println("After year " + year + " at " + INT_RATE +
            " interest rate, balance is $" + balance);
         year = year + 1;
         System.out.println("\nDo you want to see the balance " +
            " at the end of another year?");
         System.out.println("Enter 1 for yes");
         System.out.print("   or any other number for no >> ");
         response = keyboard.nextInt();
      } while(response == 1);
   }
}
```

Figure 6-21 A `do...while` loop for the `BankBalance2` application

Learning About Nested Loops

- **Inner loop** and **outer loop**
  - An inner loop must be entirely contained in an outer loop
  - Loops can never overlap
- To print three mailing labels for each of 20 customers:

```
for(customer = 1; customer <= 20; ++customer)
   for(color = 1; color <= 3; ++color)
       outputLabel ();
```

Learning About Nested Loops (cont'd.)

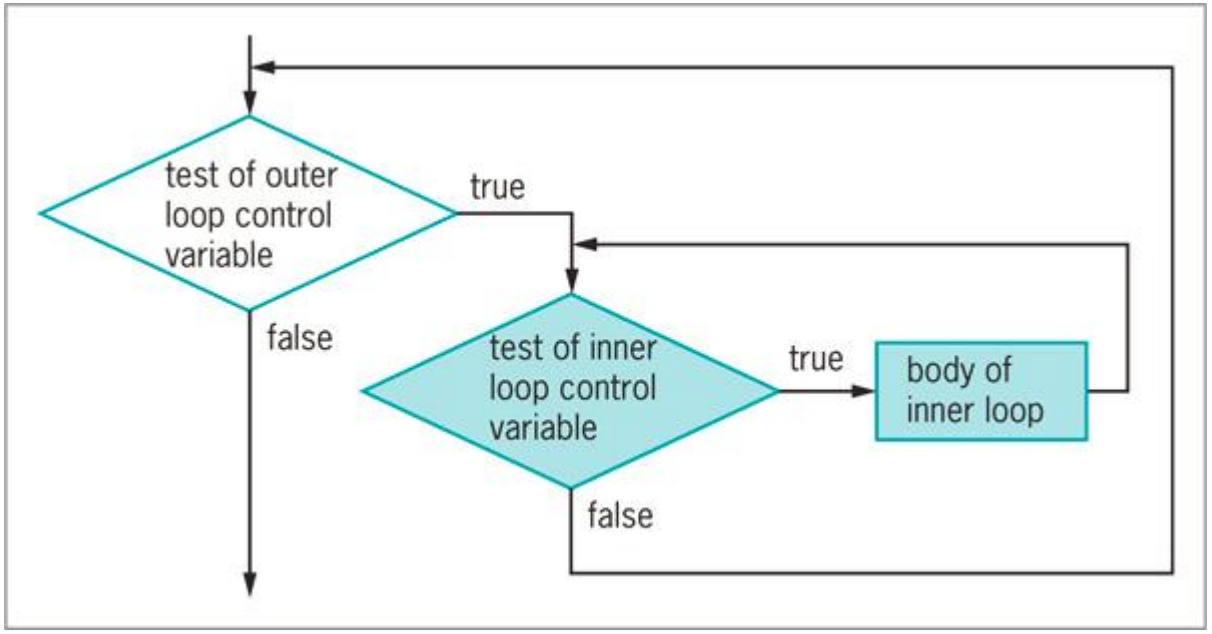


Figure 6-23 Nested loops

Improving Loop Performance

- Make sure a loop does not include unnecessary operations or statements
- Consider the order of evaluation for short-circuit operators
- Make comparisons to 0
- Employ loop fusion

Avoiding Unnecessary Operations

- Do not use unnecessary operations or statements:
  - Within a loop's tested expression
  - Within the loop body
- Avoid:

```
while (x < a + b)
// loop body
```

- Instead use:

```
int sum = a + b;
while(x < sum)
// loop body
```

Considering the Order of Evaluation of Short-Circuit Operators

- Short-circuit evaluation
  - Each part of an AND or an OR expression is evaluated only as much as necessary to determine the value of the expression
- It's important to consider the number of evaluations that take place
  - When a loop might execute many times

Comparing to Zero

- Making a comparison to 0 is faster than making a comparison to any other value
- To improve loop performance, compare the loop control variable to 0
- **Do-nothing loop**
  - Performs no actions other than looping

Comparing to Zero (cont'd.)

```
public class CompareLoops
{
   public static void main(String[] args)
   {
      long startTime1, startTime2, endTime1, endTime2;
      final int REPEAT = 100000;
      startTime1 = System.currentTimeMillis();
      for(int x = 0; x <= REPEAT; ++x)
         for(int y = 0; y <= REPEAT; ++y);
      endTime1 = System.currentTimeMillis();
      System.out.println("Time for loops starting from 0: " +
         (endTime1 - startTime1) + " milliseconds");
      startTime2 = System.currentTimeMillis();
      for(int x = REPEAT; x >= 0; --x)
         for(int y = REPEAT; y >= 0; --y);
      endTime2 = System.currentTimeMillis();
      System.out.println("Time for loops ending at 0: " +
         (endTime2 - startTime2) + " milliseconds");
   }
}
```

Figure 6-27 The `CompareLoops` application

Employing Loop Fusion

- **Loop fusion**
  - A technique of combining two loops into one
  - Will not work in every situation

Using Prefix Incrementing Rather than Postfix Incrementing

- Prefix incrementing method
  - `++x`
  - When the method receives a reference to `x`, the value is increased and the increased value is returned
- Postfix incrementing method
  - `x++`
  - When the method receives a reference to `x`, a copy of the value is made and stored
  - The value is incremented as indicated by the reference
  - The copy is returned
  - The extra time spent copying causes postfix incrementing to take longer

Using Prefix Incrementing Rather than Postfix Incrementing (cont'd.)

```
public class CompareLoops2
{
   public static void main(String[] args)
   {
      long startTime1, startTime2, endTime1, endTime2;
      final long REPEAT = 1000000000L;
      startTime1 = System.currentTimeMillis();
      for(int x = 0; x < REPEAT; x++);
      endTime1 = System.currentTimeMillis();
      System.out.println("Time with postfix increment: " +
         (endTime1 - startTime1)+ " milliseconds");
      startTime2 = System.currentTimeMillis();
      for(int x = 0; x < REPEAT; ++x);
      endTime2 = System.currentTimeMillis();
      System.out.println("Time with prefix increment: " +
         (endTime2 - startTime2)+ " milliseconds");
   }
}
```

Figure 6-29 The `CompareLoops2` program

You Do It

- Writing a Loop to Validate Data Entries
- Working with Prefix and Postfix Increment Operators
- Working with Definite Loops
- Working with Nested Loops
- Comparing Execution Times for Separate and Fused Loops

Don’t Do It

- Don’t insert a semicolon at the end of a `while` clause
- Don’t forget to block multiple statements that should execute in a loop
- Don’t make the mistake of checking for invalid data using a decision instead of a loop
- Don’t ignore subtleties in the boundaries used to stop loop performance
- Don’t repeat steps within a loop that could just as well be placed outside the loop

Summary

- The loop structure allows repeated execution of a block of statements
  - Infinite loop
  - Definite loop
  - Nest loop
- You must change the loop control variable within the looping structure
- Use the `while` loop to execute statements while some condition is `true`

Summary (cont'd.)

- Execute the `while` loop
  - Initialize the loop control variable, test in the `while` statement, and alter the loop control variable
- Prefix ++ and postfix ++
  - Increase a variable's value by 1
  - The variable is used
    - The result is calculated and stored
- Unary operators
  - Use with one value

Summary (cont'd.)

- Binary operators
  - Operate on two values
- Shortcut operators `+=`, `-=`, `*=`, and `/=`
  - Perform operations and assign the result in one step
- `for` loop
  - Initializes, tests, and increments in one statement
- `do…while` loop
  - Tests a Boolean expression after one repetition
- Improve loop performance
  - Do not include unnecessary operations or statements