Software Project

baldoino@ic.ufal.br

Java How to Program (Tenth Edition), Paul Deitel e Harvey Deitel


Java™

Introduction to Java

Major problems in the 70's

- **Pointers**
- **Memory management?**
- **Organization?**
- **Lack of libraries?**
- **Have to rewrite part of the code when changing operating system?**
- **Cost of using technology?**

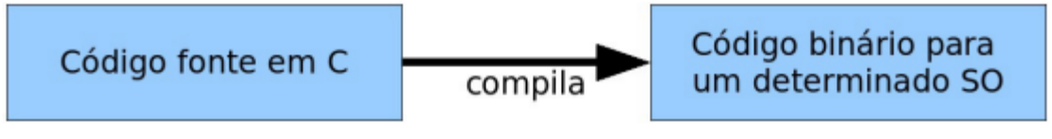
Introduction to Java

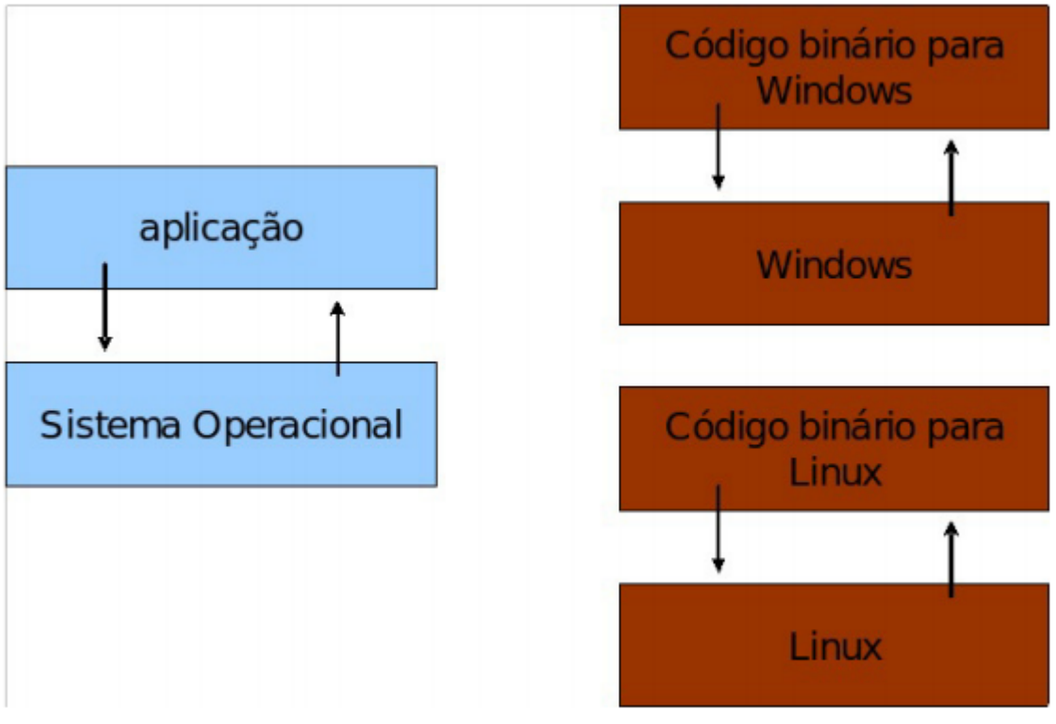
Major problems in the 70's

- **Created by sun in 1992;**
- **Initially it was created to be an interpreter for different devices (VHS,TV...)**
- **The idea did not work, and tried to use the technology to run small applications inside the browser (Applet).**
- **In the end none of them became the focus of Java.**
- **In 2009, Oracle acquired Sun.**

Introduction to Java

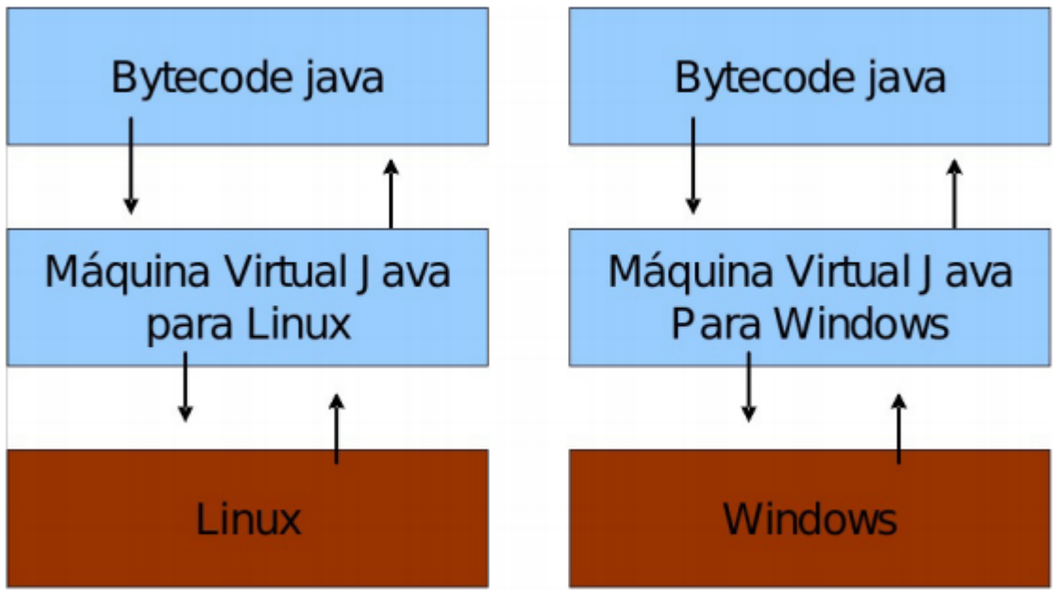
Virtual Machine





Introduction to Java

Virtual Machine



- **“WRITE ONCE, RUN ANYWHERE” (Sun)**

Introduction to Java

Versions

- **Java 1.0 … 1.1 … 1.5**
- **Java 5 … 6 … 7 … 8**
- **Java 9 … 10**
- **JRE (Java Runtime Enviroment)**
- **JDK (Java Developer Kit)**
- **The JDK and JRE can be downloaded in http://www.oracle.com/technetwork/java/**

Classes and Objects

Computer Science is...

The Science of using and processing

large amounts of information

to automate useful tasks

and learn about the world around us

A *class* is a **type** of data

An *object* is one such **piece of data***

*with associated functionality

First Program

Input/Output and Operations

Printing a Line of Text

A compilation error occurs if a public class's filename is not exactly same name as the class

```
1  // Fig. 2.1: Welcome1.java
2  // Text-printing program.
3
4  public class Welcome1
5  {
6     // main method begins execution of Java application
7     public static void main(String[] args)
8     {
9        System.out.println("Welcome to Java Programming!");
10    } // end method main
11 } // end class Welcome1
```

Use *blank lines* and *spaces* to enhance program readability

Input/Output and Operations

Executing your first program

Input/Output and Operations

Printing Text

```
1  // Fig. 2.3: Welcome2.java
2  // Printing a line of text with multiple statements.
3
4  public class Welcome2
5  {
6     // main method begins execution of Java application
7     public static void main(String[] args)
8     {
9        System.out.print("Welcome to ");
10       System.out.println("Java Programming!");
11    } // end method main
12 } // end class Welcome2
```

```
System.out.println("Welcome\nto\nJava\nProgramming!");
```

Input/Output and Operations
Another Application: Adding Integers

import Declarations

```
1  // Fig. 2.7: Addition.java
2  // Addition program that inputs two numbers then displays their sum.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class Addition
6  {
7     // main method begins execution of Java application
8     public static void main(String[] args)
9     {
10       // create a Scanner to obtain input from the command window
11       Scanner input = new Scanner(System.in);
12
13       int number1; // first number to add
14       int number2; // second number to add
15       int sum; // sum of number1 and number2
16
17       System.out.print("Enter first integer: "); // prompt
18       number1 = input.nextInt(); // read first number from user
19
20       System.out.print("Enter second integer: "); // prompt
21       number2 = input.nextInt(); // read second number from user
22
23       sum = number1 + number2; // add numbers, then store total in sum
24
25       System.out.printf("Sum is %d%n", sum); // display sum
26    } // end method main
27 } // end class Addition
```

Input/Output and Operations

Arithmetic

Java operation	Operator	Algebraic expression	Java expression
Addition	+	$f + 7$	`f + 7`
Subtraction	-	$p - c$	`p - c`
Multiplication	*	bm	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

Input/Output and Operations

Rules of Operator Precedence

Operator(s)	Operation(s)	Order of evaluation (precedence)
*	Multiplication	Evaluated first. If there are several operators of this type, they're evaluated from *left to right*.
/	Division	
%	Remainder	
+	Addition	Evaluated next. If there are several operators of this type, they're evaluated from *left to right*.
-	Subtraction	
=	Assignment	Evaluated last.

Input/Output and Operations

Equality Operators

Algebraic operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Equality operators			
=	==	x == y	x is equal to y
1	!=	x != y	x is not equal to y

Input/Output and Operations

Equality Operators

Algebraic operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Input/Output and Operations

Operators

```java
// Fig. 2.15: Comparison.java
// Compare integers using if statements, relational operators
// and equality operators.
import java.util.Scanner; // program uses class Scanner

public class Comparison
{
   // main method begins execution of Java application
   public static void main(String[] args)
   {
      // create Scanner to obtain input from command line
      Scanner input = new Scanner(System.in);

      int number1; // first number to compare
      int number2; // second number to compare

      System.out.print("Enter first integer: "); // prompt
      number1 = input.nextInt(); // read first number from user

      System.out.print("Enter second integer: "); // prompt
      number2 = input.nextInt(); // read second number from user

      if (number1 == number2)
         System.out.printf("%d == %d%n", number1, number2);

      if (number1 != number2)
         System.out.printf("%d != %d%n", number1, number2);

      if (number1 < number2)
         System.out.printf("%d < %d%n", number1, number2);

      if (number1 > number2)
         System.out.printf("%d > %d%n", number1, number2);

      if (number1 <= number2)
         System.out.printf("%d <= %d%n", number1, number2);

      if (number1 >= number2)
         System.out.printf("%d >= %d%n", number1, number2);
   } // end method main
} // end class Comparison
```

Format printing

%d	representa números inteiros
%f	representa números floats
%2f	representa números doubles
%b	representa valores booleanos
%c	representa valores char

Input/Output and Operations

Precedence and Associativity of Operators

Operators				Associativity	Type
*	/	%		left to right	multiplicative
+	-			left to right	additive
<	<=	>	>=	left to right	relational
==	!=			left to right	equality
=				right to left	assignment

Introduction to Classes

Introduction to Classes

Instance Variable, *set* and *get*

Implement a Class Account!

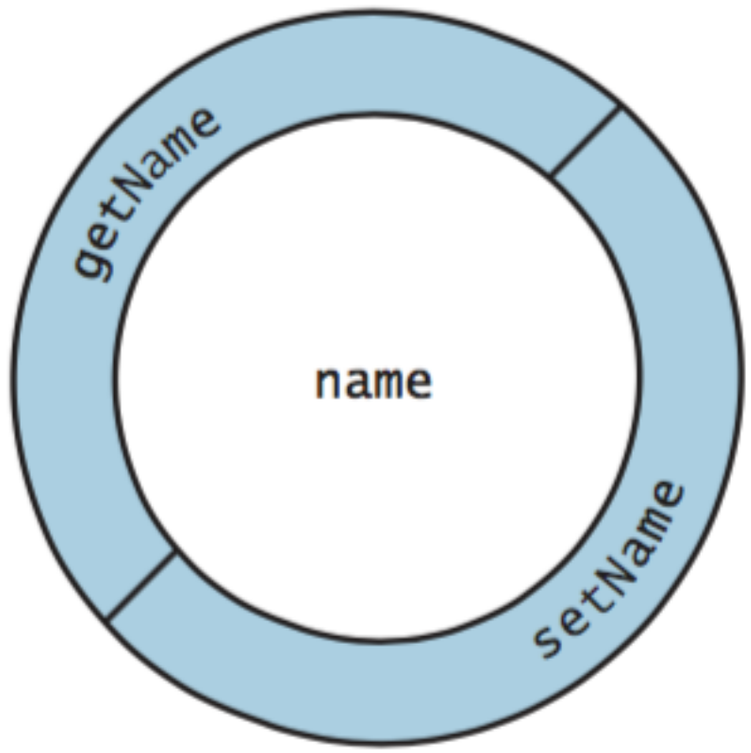
Introduction to Classes

Instance Variable, *set* and *get*

```
1  // Fig. 3.1: Account.java
2  // Account class that contains a name instance variable
3  // and methods to set and get its value.
4
5  public class Account
6  {
7     private String name; // instance variable
8
9     // method to set the name in the object
10    public void setName(String name)
11    {
12       this.name = name; // store the name
13    }
14
15    // method to retrieve the name from the object
16    public String getName()
17    {
18       return name; // return value of name to caller
19    }
20 } // end class Account
```

Introduction to Classes

Instance Variable, *set* and *get*



Account Object with Encapsuled Data

Introduction to Classes

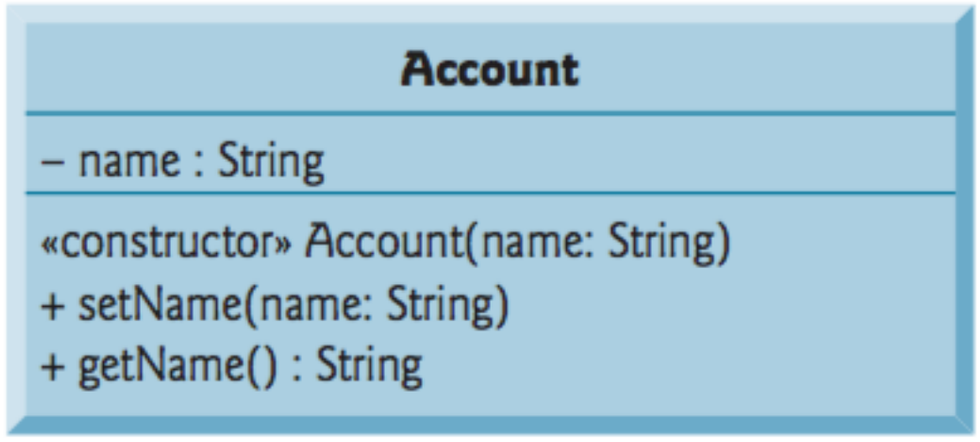
Using an Object of Class *Account*

```
1  // Fig. 3.2: AccountTest.java
2  // Creating and manipulating an Account object.
3  import java.util.Scanner;
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        // create a Scanner object to obtain input from the command window
10       Scanner input = new Scanner(System.in);
11
12       // create an Account object and assign it to myAccount
13       Account myAccount = new Account();
14
15       // display initial value of name (null)
16       System.out.printf("Initial name is: %s%n%n", myAccount.getName());
17
18       // prompt for and read name
19       System.out.println("Please enter the name:");
20       String theName = input.nextLine(); // read a line of text
21       myAccount.setName(theName); // put theName in myAccount
22       System.out.println(); // outputs a blank line
23
24       // display the name stored in object myAccount
25       System.out.printf("Name in object myAccount is:%n%s%n",
26          myAccount.getName());
27    }
28 } // end class AccountTest
```

Introduction to Classes
Initialising Objects with Constructors

```
1  // Fig. 3.5: Account.java
2  // Account class with a constructor that initializes the name.
3
4  public class Account
5  {
6     private String name; // instance variable
7
8     // constructor initializes name with parameter name
9     public Account(String name) // constructor name is class name
10    {
11       this.name = name;
12    }
13
14    // method to set the name
15    public void setName(String name)
16    {
17       this.name = name;
18    }
19
20    // method to retrieve the name
21    public String getName()
22    {
23       return name;
24    }
25 } // end class Account
```

Introduction to Classes
Initialising Objects with Constructors



Upgrade with a Constructor!!

Introduction to Classes

Initialising *Account* objects when They're Created

```
1  // Fig. 3.6: AccountTest.java
2  // Using the Account constructor to initialize the name instance
3  // variable at the time each Account object is created.
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        // create two Account objects
10       Account account1 = new Account("Jane Green");
11       Account account2 = new Account("John Blue");
12
13       // display initial value of name for each Account
14       System.out.printf("account1 name is: %s%n", account1.getName());
15       System.out.printf("account2 name is: %s%n", account2.getName());
16    }
17 } // end class AccountTest
```

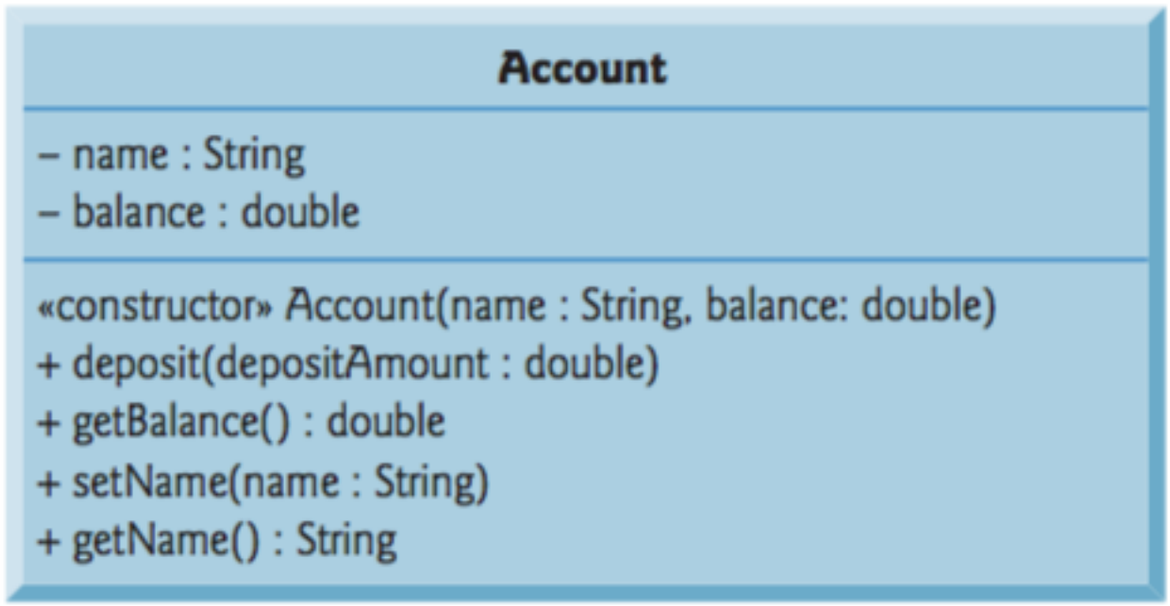
Introduction to Classes

balance Instance Variable

```
1  // Fig. 3.8: Account.java
2  // Account class with a double instance variable balance and a constructor
3  // and deposit method that perform validation.
4
5  public class Account
6  {
7     private String name; // instance variable
8     private double balance; // instance variable
9
10    // Account constructor that receives two parameters
11    public Account(String name, double balance)
12    {
13       this.name = name; // assign name to instance variable name
14
15       // validate that the balance is greater than 0.0; if it's not,
16       // instance variable balance keeps its default initial value of 0.0
17       if (balance > 0.0) // if the balance is valid
18          this.balance = balance; // assign it to instance variable balance
19    }
20
21    // method that deposits (adds) only a valid amount to the balance
22    public void deposit(double depositAmount)
23    {
24       if (depositAmount > 0.0) // if the depositAmount is valid
25          balance = balance + depositAmount; // add it to the balance
26    }
27
28    // method returns the account balance
29    public double getBalance()
30    {
31       return balance;
32    }
33
34    // method that sets the name
35    public void setName(String name)
36    {
37       this.name = name;
38    }
39
40    // method that returns the name
41    public String getName()
42    {
43       return name; // give value of name back to caller
44    } // end method getName
45 } // end class Account
```

Introduction to Classes

***balance* Instance Variable**



Keep Evolving!!

Introduction to Classes

AccountTest* class to use class *Account

```
1  // Fig. 3.9: AccountTest.java
2  // Inputting and outputting floating-point numbers with Account objects.
3  import java.util.Scanner;
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        Account account1 = new Account("Jane Green", 50.00);
10       Account account2 = new Account("John Blue", -7.53);
11
12       // display initial balance of each object
13       System.out.printf("%s balance: $%.2f%n",
14          account1.getName(), account1.getBalance());
15       System.out.printf("%s balance: $%.2f%n%n",
16          account2.getName(), account2.getBalance());
17
18       // create a Scanner to obtain input from the command window
19       Scanner input = new Scanner(System.in);
20
21       System.out.print("Enter deposit amount for account1: "); // prompt
22       double depositAmount = input.nextDouble(); // obtain user input
23       System.out.printf("%nadding %.2f to account1 balance%n%n",
24          depositAmount);
25       account1.deposit(depositAmount); // add to account1's balance
26
27       // display balances
28       System.out.printf("%s balance: $%.2f%n",
29          account1.getName(), account1.getBalance());
30       System.out.printf("%s balance: $%.2f%n%n",
31          account2.getName(), account2.getBalance());
32
33       System.out.print("Enter deposit amount for account2: "); // prompt
34       depositAmount = input.nextDouble(); // obtain user input
35       System.out.printf("%nadding %.2f to account2 balance%n%n",
36          depositAmount);
37       account2.deposit(depositAmount); // add to account2 balance
38
39       // display balances
40       System.out.printf("%s balance: $%.2f%n",
41          account1.getName(), account1.getBalance());
42       System.out.printf("%s balance: $%.2f%n%n",
43          account2.getName(), account2.getBalance());
44    } // end main
45 } // end class AccountTest
```

Introduction to Classes

Removing Duplicated Code in Method main

“In the AccountTest, the main method contains six statements (lines 13–14, 15–16, 28–29, 30–31, 40–41 and 42–43) that each display an Account object’s name and balance.”

Introduction to Classes

Removing Duplicated Code in Method main

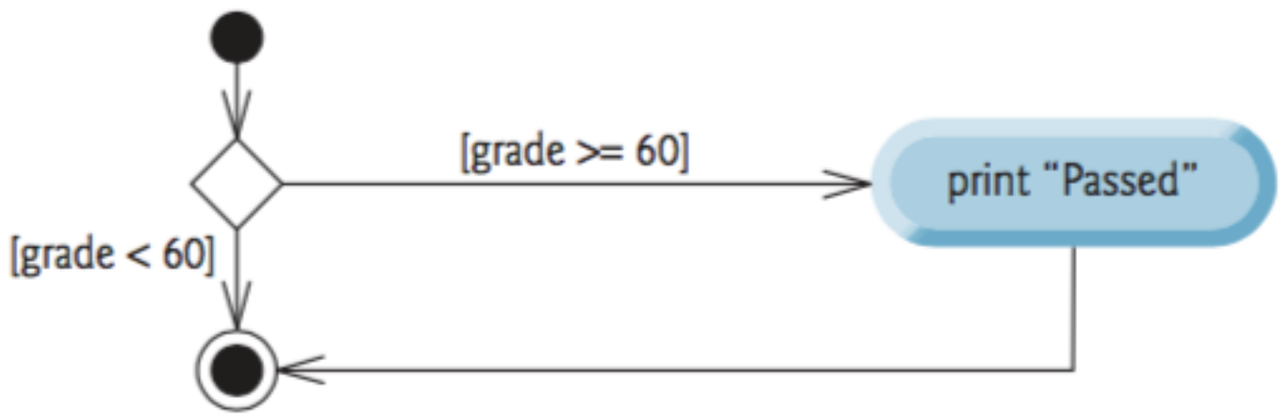
```
public static void displayAccount(Account accountToDisplay)
{
   // place the statement that displays
   // accountToDisplay's name and balance here
}
```

Assignment, ++ and - - Operators

Control Structures

If student's grade is greater than or equal to 60
*   Print "Passed"*

```
if (studentGrade >= 60)
   System.out.println("Passed");
```

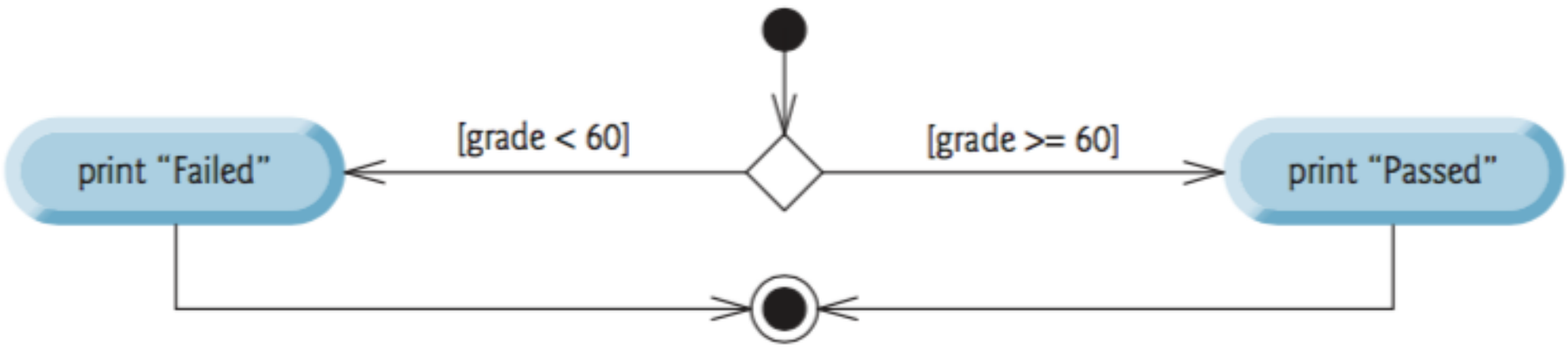


Assignment, ++ and - - Operators

if...else Double-Selection Statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
Else
    Print "Failed"
```

```
if (grade >= 60)
   System.out.println("Passed");
else
   System.out.println("Failed");
```



Assignment, ++ and - - Operators

Nested if…else Statements

```
If student's grade is greater than or equal to 90
        Print "A"
else
        If student's grade is greater than or equal to 80
                Print "B"
        else
                If student's grade is greater than or equal to 70
                        Print "C"
                else
                        If student's grade is greater than or equal to 60
                                Print "D"
                        else
                                Print "F"
```

Assignment, ++ and - - Operators

Nested if...else Statements

```
if (studentGrade >= 90)
   System.out.println("A");
else if (studentGrade >= 80)
   System.out.println("B");
else if (studentGrade >= 70)
   System.out.println("C");
else if (studentGrade >= 60)
   System.out.println("D");
else
   System.out.println("F");
```

Assignment, ++ and - - Operators

Dangling-else Problem

```
if (x > 5)
   if (y > 5)
      System.out.println("x and y are > 5");
else
   System.out.println("x is <= 5");
```

```
if (x > 5)
{
   if (y > 5)
      System.out.println("x and y are > 5");
}
else
   System.out.println("x is <= 5");
```

```
if (x > 5)
   if (y > 5)
      System.out.println("x and y are > 5");
   else
      System.out.println("x is <= 5");
```

Assignment, ++ and - - Operators

Conditional Operator (?:)

```
System.out.println(studentGrade >= 60 ? "Passed" : "Failed");
```

Assignment, ++ and - - Operators

```
1  // Fig. 4.4: Student.java
2  // Student class that stores a student name and average.
3  public class Student
4  {
5     private String name;
6     private double average;
7
8     // constructor initializes instance variables
9     public Student(String name, double average)
10    {
11       this.name = name;
12
13       // validate that average is > 0.0 and <= 100.0; otherwise,
14       // keep instance variable average's default value (0.0)
15       if (average > 0.0)
16          if (average <= 100.0)
17             this.average = average; // assign to instance variable
18    }
19
20    // sets the Student's name
21    public void setName(String name)
22    {
23       this.name = name;
24    }
25
26    // retrieves the Student's name
27    public String getName()
28    {
29       return name;
30    }
31
32    // sets the Student's average
33    public void setAverage(double studentAverage)
34    {
35       // validate that average is > 0.0 and <= 100.0; otherwise,
36       // keep instance variable average's current value
37       if (average > 0.0)
38          if (average <= 100.0)
39             this.average = average; // assign to instance variable
40    }
41
42    // retrieves the Student's average
43    public double getAverage()
44    {
45       return average;
46    }
47
48    // determines and returns the Student's letter grade
49    public String getLetterGrade()
50    {
51       String letterGrade = ""; // initialized to empty String
52
53       if (average >= 90.0)
54          letterGrade = "A";
55       else if (average >= 80.0)
56          letterGrade = "B";
57       else if (average >= 70.0)
58          letterGrade = "C";
59       else if (average >= 60.0)
60          letterGrade = "D";
61       else
62          letterGrade = "F";
63
64       return letterGrade;
65    }
66 } // end class Student
```

Assignment, ++ and - - Operators

StudentTest class

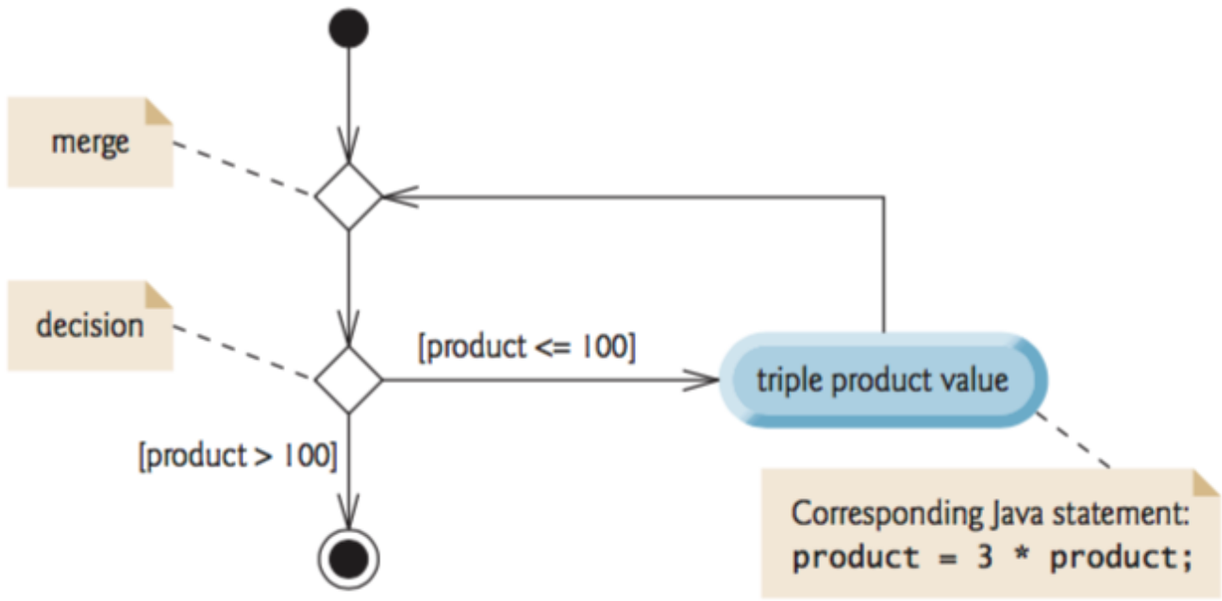
```
1  // Fig. 4.5: StudentTest.java
2  // Create and test Student objects.
3  public class StudentTest
4  {
5     public static void main(String[] args)
6     {
7        Student account1 = new Student("Jane Green", 93.5);
8        Student account2 = new Student("John Blue", 72.75);
9
10       System.out.printf("%s's letter grade is: %s%n",
11          account1.getName(), account1.getLetterGrade());
12       System.out.printf("%s's letter grade is: %s%n",
13          account2.getName(), account2.getLetterGrade());
14    }
15 } // end class StudentTest
```

Assignment, ++ and - - Operators

***while* Repetition Statement**

While there are more items on my shopping list
    Purchase next item and cross it off my list

```
while (product <= 100)
   product = 3 * product;
```



Assignment, Operators ++ and - -

Control Variable Repetition

```
1    Set total to zero
2    Set grade counter to one
3
4    While grade counter is less than or equal to ten
5        Prompt the user to enter the next grade
6        Input the next grade
7        Add the grade into the total
8        Add one to the grade counter
9
10   Set the class average to the total divided by ten
11   Print the class average
```

Assignment, ++ and - - Operators

Control Variable Repetition

```
1  // Fig. 4.8: ClassAverage.java
2  // Solving the class-average problem using counter-controlled repetition.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class ClassAverage
6  {
7     public static void main(String[] args)
8     {
9        // create Scanner to obtain input from command window
10       Scanner input = new Scanner(System.in);
11
12       // initialization phase
13       int total = 0;  // initialize sum of grades entered by the user
14       int gradeCounter = 1; // initialize # of grade to be entered next
15
16       // processing phase uses counter-controlled repetition
17       while (gradeCounter <= 10) // loop 10 times
18       {
19          System.out.print("Enter grade: "); // prompt
20          int grade = input.nextInt(); // input next grade
21          total = total + grade; // add grade to total
22          gradeCounter = gradeCounter + 1; // increment counter by 1
23       }
24
25       // termination phase
26       int average = total / 10; // integer division yields integer result
27
28       // display total and average of grades
29       System.out.printf("%nTotal of all 10 grades is %d%n", total);
30       System.out.printf("Class average is %d%n", average);
31    }
32 } // end class ClassAverage
```

Assignment, ++ and - - Operators

Compound Assignment Operators

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Assignment, ++ and - - Operators

Increment and Decrement Operators

Operator	Operator name	Sample expression	Explanation
`++`	prefix increment	`++a`	Increment `a` by 1, then use the new value of `a` in the expression in which `a` resides.
`++`	postfix increment	`a++`	Use the current value of `a` in the expression in which `a` resides, then increment `a` by 1.
`--`	prefix decrement	`--b`	Decrement `b` by 1, then use the new value of `b` in the expression in which `b` resides.
`--`	postfix decrement	`b--`	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by 1.

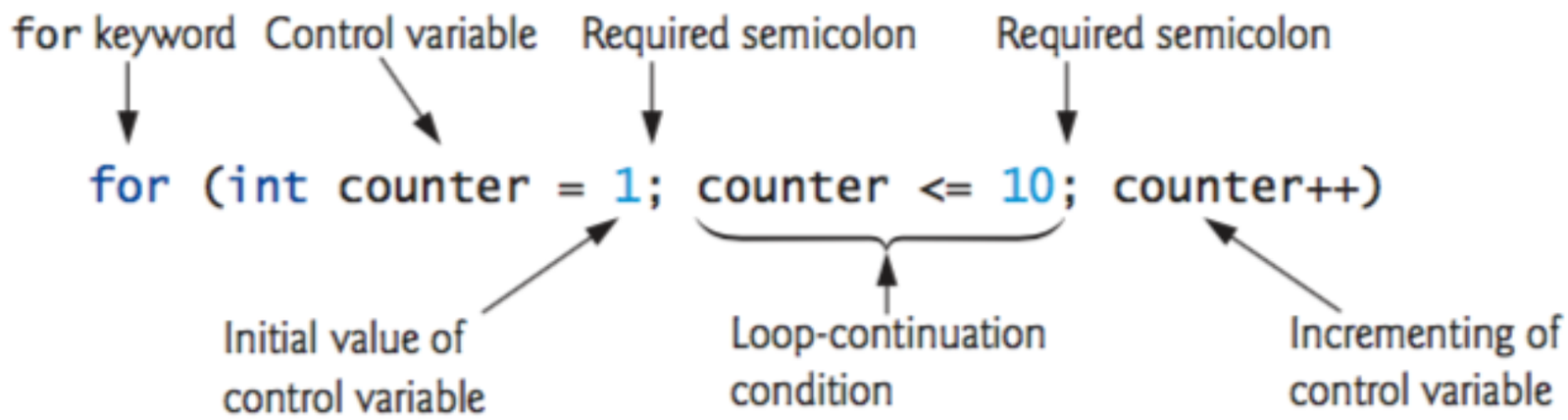
Assignment, ++ and - - Operators

Prefix and Postfix Increment

```
1  // Fig. 4.15: Increment.java
2  // Prefix increment and postfix increment operators.
3
4  public class Increment
5  {
6     public static void main(String[] args)
7     {
8        // demonstrate postfix increment operator
9        int c = 5;
10       System.out.printf("c before postincrement: %d%n", c); // prints 5
11       System.out.printf("    postincrementing c: %d%n", c++); // prints 5
12       System.out.printf(" c after postincrement: %d%n", c); // prints 6
13
14       System.out.println(); // skip a line
15
16       // demonstrate prefix increment operator
17       c = 5;
18       System.out.printf(" c before preincrement: %d%n", c); // prints 5
19       System.out.printf("     preincrementing c: %d%n", ++c); // prints 6
20       System.out.printf("  c after preincrement: %d%n", c); // prints 6
21    }
22 } // end class Increment
```

Logical Operators

***for* Repetition Statement**



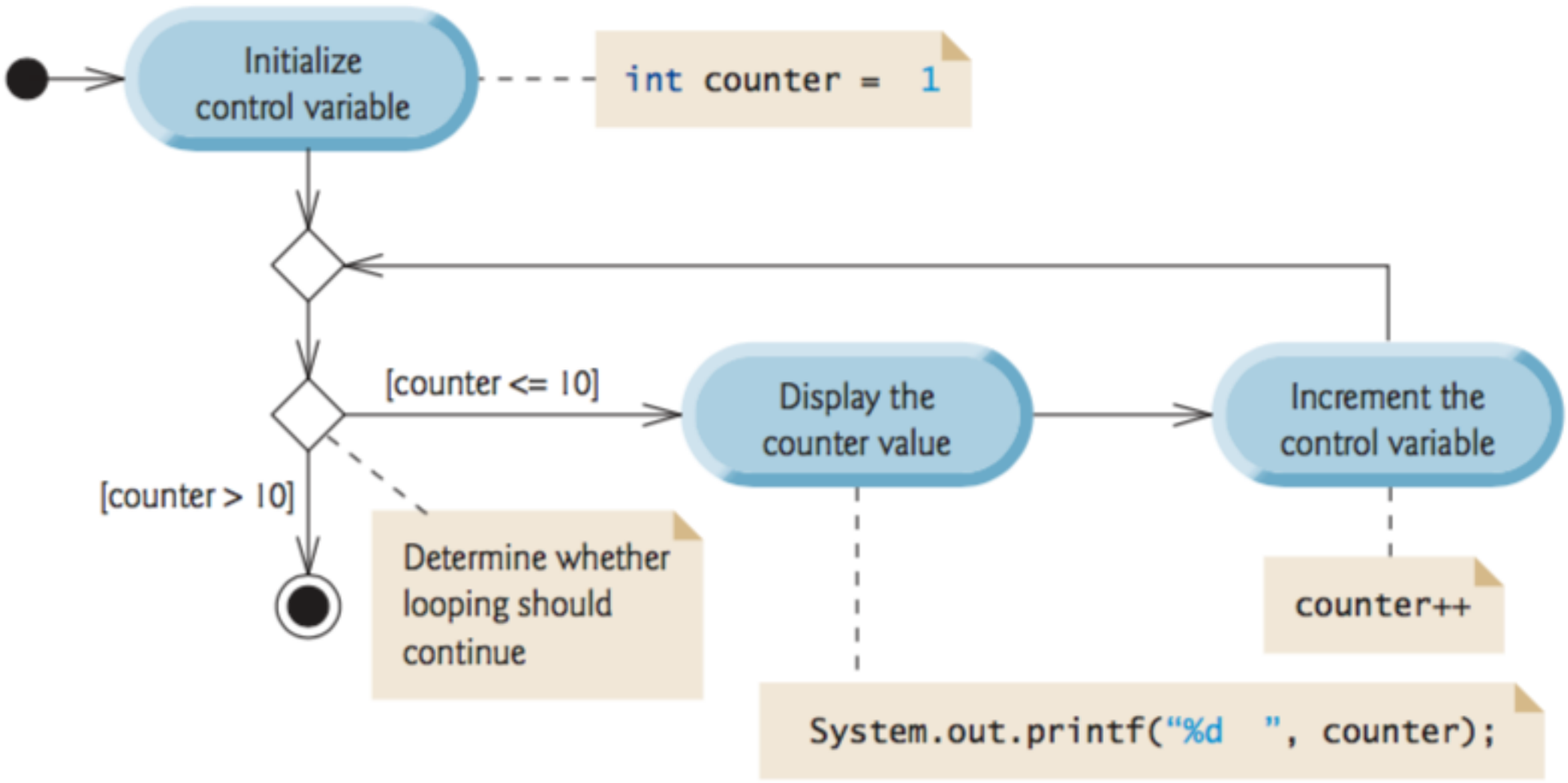
Logical Operators

for Repetition Statement

```
1  // Fig. 5.2: ForCounter.java
2  // Counter-controlled repetition with the for repetition statement.
3
4  public class ForCounter
5  {
6     public static void main(String[] args)
7     {
8        // for statement header includes initialization,
9        // loop-continuation condition and increment
10       for (int counter = 1; counter <= 10; counter++)
11          System.out.printf("%d  ", counter);
12
13       System.out.println();
14    }
15 } // end class ForCounter
```

Logical Operators

***for* Repetition Statement**



Logical Operators

for Repetition Statement

a) Vary the control variable from 1 to 100 in increments of 1.

```
for (int i = 1; i <= 100; i++)
```

b) Vary the control variable from 100 to 1 in *decrements* of 1.

```
for (int i = 100; i >= 1; i--)
```

c) Vary the control variable from 7 to 77 in increments of 7.

```
for (int i = 7; i <= 77; i += 7)
```

d) Vary the control variable from 20 to 2 in *decrements* of 2.

```
for (int i = 20; i >= 2; i -= 2)
```

e) Vary the control variable over the values 2, 5, 8, 11, 14, 17, 20.

```
for (int i = 2; i <= 20; i += 3)
```

f) Vary the control variable over the values 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for (int i = 99; i >= 0; i -= 11)
```

Logical Operators

***for* Repetition Statement**

```
for (int j = x; j <= 4 * x * y; j += y / x)
```

Assume that x=2 and y=10. Are the statements equivalents?

```
for (int j = 2; j <= 80; j += 5)
```

Logical Operators

***do…while* Repetition Statement**

```
do
{
    statement
} while (condition);
```

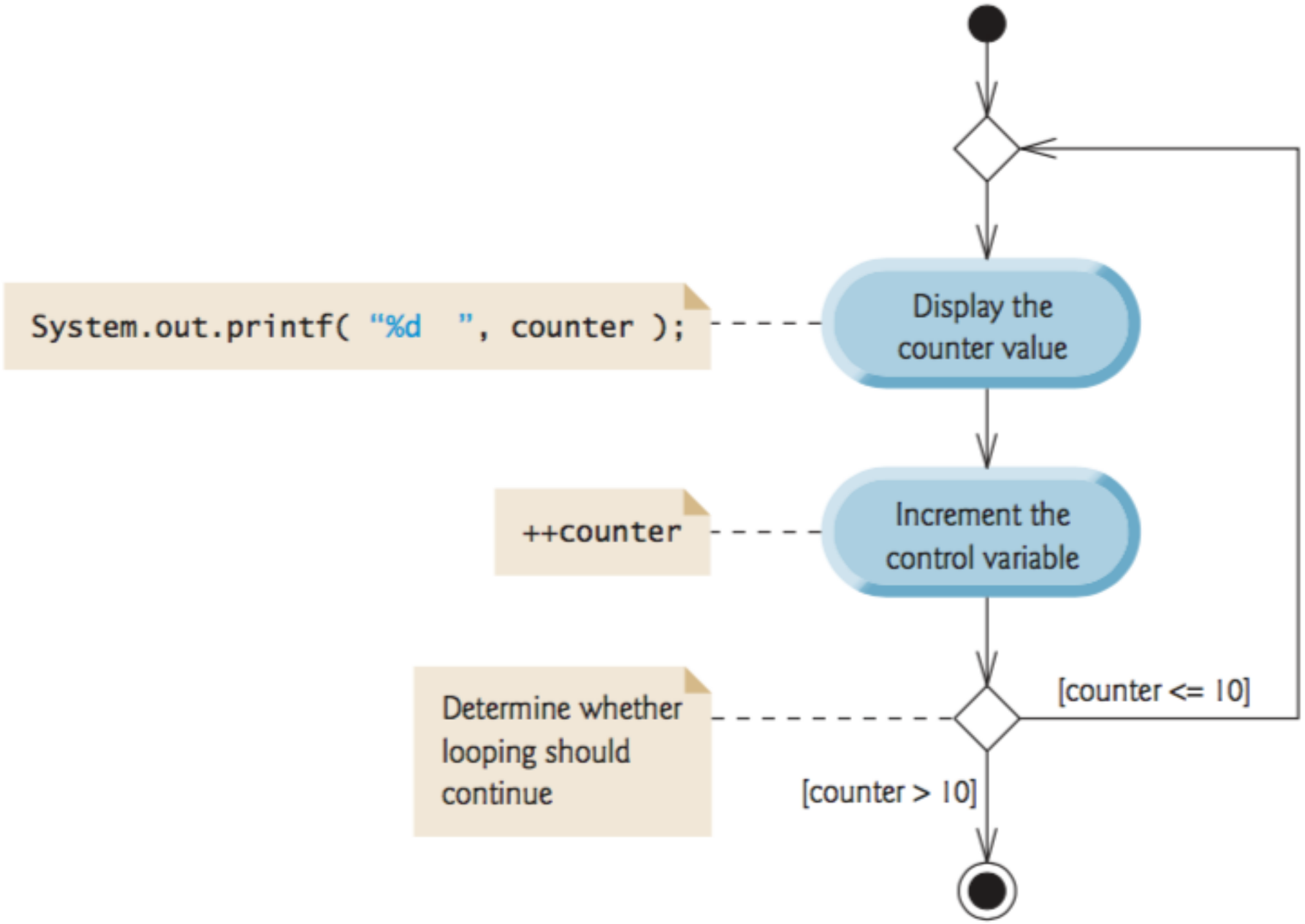
Logical Operators

do…while Repetition Statement

```
1  // Fig. 5.7: DoWhileTest.java
2  // do...while repetition statement.
3
4  public class DoWhileTest
5  {
6     public static void main(String[] args)
7     {
8        int counter = 1;
9
10       do
11       {
12          System.out.printf("%d  ", counter);
13          ++counter;
14       } while (counter <= 10); // end do...while
15
16       System.out.println();
17    }
18 } // end class DoWhileTest
```

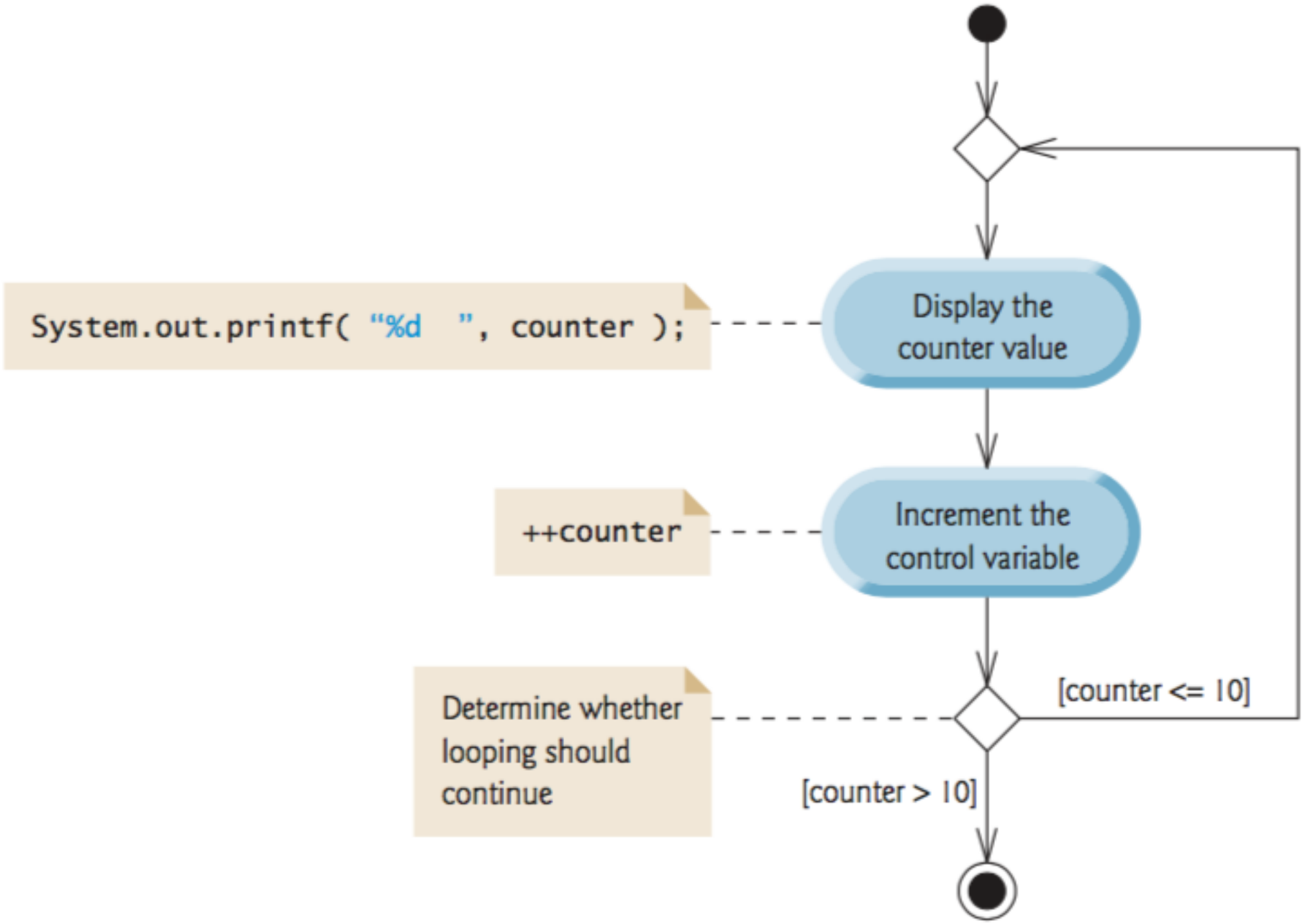
Logical Operators

***do…while* Repetition Statement**



Logical Operators

switch Multiple-Selection Statement



```java
// Fig. 5.9: LetterGrades.java
// LetterGrades class uses the switch statement to count letter grades.
import java.util.Scanner;

public class LetterGrades
{
   public static void main(String[] args)
   {
      int total = 0; // sum of grades
      int gradeCounter = 0; // number of grades entered
      int aCount = 0; // count of A grades
      int bCount = 0; // count of B grades
      int cCount = 0; // count of C grades
      int dCount = 0; // count of D grades
      int fCount = 0; // count of F grades

      Scanner input = new Scanner(System.in);

      System.out.printf("%s%n%s%n   %s%n   %s%n",
         "Enter the integer grades in the range 0-100.",
         "Type the end-of-file indicator to terminate input:",
         "On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter",
         "On Windows type <Ctrl> z then press Enter");

      // loop until user enters the end-of-file indicator
      while (input.hasNext())
      {
         int grade = input.nextInt(); // read grade
         total += grade; // add grade to total
         ++gradeCounter; // increment number of grades

         //  increment appropriate letter-grade counter
         switch (grade / 10)
         {
            case 9:  // grade was between 90
            case 10: // and 100, inclusive
               ++aCount;
               break; // exits switch

            case 8: // grade was between 80 and 89
               ++bCount;
               break; // exits switch

            case 7: // grade was between 70 and 79
               ++cCount;
               break; // exits switch

            case 6: // grade was between 60 and 69
               ++dCount;
               break; // exits switch

            default: // grade was less than 60
               ++fCount;
               break; // optional; exits switch anyway
         } // end switch
      } // end while

      // display grade report
      System.out.printf("%nGrade Report:%n");

      // if user entered at least one grade...
      if (gradeCounter != 0)
      {
         // calculate average of all grades entered
         double average = (double) total / gradeCounter;

         // output summary of results
         System.out.printf("Total of the %d grades entered is %d%n",
            gradeCounter, total);
         System.out.printf("Class average is %.2f%n", average);
         System.out.printf("%n%s%n%s%d%n%s%d%n%s%d%n%s%d%n%s%d%n",
            "Number of students who received each grade:",
            "A: ", aCount,   // display number of A grades
            "B: ", bCount,   // display number of B grades
            "C: ", cCount,   // display number of C grades
            "D: ", dCount,   // display number of D grades
            "F: ", fCount); // display number of F grades
      } // end if
      else // no grades were entered, so output appropriate message
         System.out.println("No grades were entered");
   } // end main
} // end class LetterGrades
```

Logical Operators

switch Multiple-Selection Statement

```
1  // Fig. 5.11: AutoPolicy.java
2  // Class that represents an auto insurance policy.
3  public class AutoPolicy
4  {
5     private int accountNumber; // policy account number
6     private String makeAndModel; // car that the policy applies to
7     private String state; // two-letter state abbreviation
8
9     // constructor
10    public AutoPolicy(int accountNumber, String makeAndModel, String state)
11    {
12       this.accountNumber = accountNumber;
13       this.makeAndModel = makeAndModel;
14       this.state = state;
15    }
16
17    // sets the accountNumber
18    public void setAccountNumber(int accountNumber)
19    {
20       this.accountNumber = accountNumber;
21    }
22
23    // returns the accountNumber
24    public int getAccountNumber()
25    {
26       return accountNumber;
27    }
28
29    // sets the makeAndModel
30    public void setMakeAndModel(String makeAndModel)
31    {
32       this.makeAndModel = makeAndModel;
33    }
34
35    // returns the makeAndModel
36    public String getMakeAndModel()
37    {
38       return makeAndModel;
39    }
40
41    // sets the state
42    public void setState(String state)
43    {
44       this.state = state;
45    }
46
47    // returns the state
48    public String getState()
49    {
50       return state;
51    }
52
53    // predicate method returns whether the state has no-fault insurance
54    public boolean isNoFaultState()
55    {
56       boolean noFaultState;
57
58       // determine whether state has no-fault auto insurance
59       switch (getState()) // get AutoPolicy object's state abbreviation
60       {
61          case "MA": case "NJ": case "NY": case "PA":
62             noFaultState = true;
63             break;
64          default:
65             noFaultState = false;
66             break;
67       }
68
69       return noFaultState;
70    }
71 } // end class AutoPolicy
```

Logical Operators

switch Multiple-Selection Statement

```
1  // Fig. 5.12: AutoPolicyTest.java
2  // Demonstrating Strings in switch.
3  public class AutoPolicyTest
4  {
5     public static void main(String[] args)
6     {
7        // create two AutoPolicy objects
8        AutoPolicy policy1 =
9           new AutoPolicy(11111111, "Toyota Camry", "NJ");
10       AutoPolicy policy2 =
11          new AutoPolicy(22222222, "Ford Fusion", "ME");
12
13       // display whether each policy is in a no-fault state
14       policyInNoFaultState(policy1);
15       policyInNoFaultState(policy2);
16    }
17
18    // method that displays whether an AutoPolicy
19    // is in a state with no-fault auto insurance
20    public static void policyInNoFaultState(AutoPolicy policy)
21    {
22       System.out.println("The auto policy:");
23       System.out.printf(
24          "Account #: %d; Car: %s; State %s %s a no-fault state%n%n",
25          policy.getAccountNumber(), policy.getMakeAndModel(),
26          policy.getState(),
27          (policy.isNoFaultState() ? "is": "is not"));
28    }
29 } // end class AutoPolicyTest
```

Logical Operators

***break* Statement**

```
1  // Fig. 5.13: BreakTest.java
2  // break statement exiting a for statement.
3  public class BreakTest
4  {
5     public static void main(String[] args)
6     {
7        int count; // control variable also used after loop terminates
8
9        for (count = 1; count <= 10; count++) // loop 10 times
10       {
11          if (count == 5)
12             break; // terminates loop if count is 5
13
14          System.out.printf("%d ", count);
15       }
16
17       System.out.printf("%nBroke out of loop at count = %d%n", count);
18    }
19 } // end class BreakTest
```

Logical Operators

***continue* Statement**

```
1  // Fig. 5.14: ContinueTest.java
2  // continue statement terminating an iteration of a for statement.
3  public class ContinueTest
4  {
5     public static void main(String[] args)
6     {
7        for (int count = 1; count <= 10; count++) // loop 10 times
8        {
9           if (count == 5)
10             continue; // skip remaining code in loop body if count is 5
11
12          System.out.printf("%d ", count);
13       }
14
15       System.out.printf("%nUsed continue to skip printing 5%n");
16    }
17 } // end class ContinueTest
```

Methods: A Deeper Look

static Methods and Class Math

ClassName.methodName(arguments)

```
Math.sqrt(900.0)
```

Method	Description	Example
abs(*x*)	absolute value of *x*	abs(23.7) is 23.7 abs(0.0) is 0.0 abs(-23.7) is 23.7
ceil(*x*)	rounds *x* to the smallest integer not less than *x*	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(*x*)	trigonometric cosine of *x* (*x* in radians)	cos(0.0) is 1.0
exp(*x*)	exponential method e^x	exp(1.0) is 2.71828 exp(2.0) is 7.38906
floor(*x*)	rounds *x* to the largest integer not greater than *x*	floor(9.2) is 9.0 floor(-9.8) is -10.0
log(*x*)	natural logarithm of *x* (base *e*)	log(Math.E) is 1.0 log(Math.E * Math.E) is 2.0
max(*x*, *y*)	larger value of *x* and *y*	max(2.3, 12.7) is 12.7 max(-2.3, -12.7) is -2.3
min(*x*, *y*)	smaller value of *x* and *y*	min(2.3, 12.7) is 2.3 min(-2.3, -12.7) is -12.7
pow(*x*, *y*)	*x* raised to the power *y* (i.e., x^y)	pow(2.0, 7.0) is 128.0 pow(9.0, 0.5) is 3.0
sin(*x*)	trigonometric sine of *x* (*x* in radians)	sin(0.0) is 0.0
sqrt(*x*)	square root of *x*	sqrt(900.0) is 30.0
tan(*x*)	trigonometric tangent of *x* (*x* in radians)	tan(0.0) is 0.0

Methods: A Deeper Look

Casting

Type	Valid promotions
`double`	None
`float`	`double`
`long`	`float` or `double`
`int`	`long`, `float` or `double`
`char`	`int`, `long`, `float` or `double`
`short`	`int`, `long`, `float` or `double` (but not `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (but not `char`)
`boolean`	None (`boolean` values are not considered to be numbers in Java)

Methods: A Deeper Look

Enum types

```
1  // Fig. 6.8: Craps.java
2  // Craps class simulates the dice game craps.
3  import java.security.SecureRandom;
4
5  public class Craps
6  {
7     // create secure random number generator for use in method rollDice
8     private static final SecureRandom randomNumbers = new SecureRandom();
9
10    // enum type with constants that represent the game status
11    private enum Status { CONTINUE, WON, LOST };
12
13    // constants that represent common rolls of the dice
14    private static final int SNAKE_EYES = 2;
15    private static final int TREY = 3;
16    private static final int SEVEN = 7;
17    private static final int YO_LEVEN = 11;
18    private static final int BOX_CARS = 12;
19
20    // plays one game of craps
21    public static void main(String[] args)
22    {
23       int myPoint = 0; // point if no win or loss on first roll
24       Status gameStatus; // can contain CONTINUE, WON or LOST
25
26       int sumOfDice = rollDice(); // first roll of the dice
27
28       // determine game status and point based on first roll
29       switch (sumOfDice)
30       {
31          case SEVEN: // win with 7 on first roll
32          case YO_LEVEN: // win with 11 on first roll
33             gameStatus = Status.WON;
34             break;
35          case SNAKE_EYES: // lose with 2 on first roll
36          case TREY: // lose with 3 on first roll
37          case BOX_CARS: // lose with 12 on first roll
38             gameStatus = Status.LOST;
39             break;
40          default: // did not win or lose, so remember point
41             gameStatus = Status.CONTINUE; // game is not over
42             myPoint = sumOfDice; // remember the point
43             System.out.printf("Point is %d%n", myPoint);
44             break;
45       }
46
47       // while game is not complete
48       while (gameStatus == Status.CONTINUE) // not WON or LOST
49       {
50          sumOfDice = rollDice(); // roll dice again
51
52          // determine game status
53          if (sumOfDice == myPoint) // win by making point
54             gameStatus = Status.WON;
55          else
56             if (sumOfDice == SEVEN) // lose by rolling 7 before point
57                gameStatus = Status.LOST;
58       }
59
```

Methods: A Deeper Look

Scope of Declarations

```
1  // Fig. 6.9: Scope.java
2  // Scope class demonstrates field and local variable scopes.
3
4  public class Scope
5  {
6     // field that is accessible to all methods of this class
7     private static int x = 1;
8
9     // method main creates and initializes local variable x
10    // and calls methods useLocalVariable and useField
11    public static void main(String[] args)
12    {
13       int x = 5; // method's local variable x shadows field x
14
15       System.out.printf("local x in main is %d%n", x);
16
17       useLocalVariable(); // useLocalVariable has local x
18       useField(); // useField uses class Scope's field x
19       useLocalVariable(); // useLocalVariable reinitializes local x
20       useField(); // class Scope's field x retains its value
21
22       System.out.printf("%nlocal x in main is %d%n", x);
23    }
24
25    // create and initialize local variable x during each call
26    public static void useLocalVariable()
27    {
28       int x = 25; // initialized each time useLocalVariable is called
29
30       System.out.printf(
31          "%nlocal x on entering method useLocalVariable is %d%n", x);
32       ++x; // modifies this method's local variable x
33       System.out.printf(
34          "local x before exiting method useLocalVariable is %d%n", x);
35    }
36
37    // modify class Scope's field x during each call
38    public static void useField()
39    {
40       System.out.printf(
41          "%nfield x on entering method useField is %d%n", x);
42       x *= 10; // modifies class Scope's field x
43       System.out.printf(
44          "field x before exiting method useField is %d%n", x);
45    }
46 } // end class Scope
```

Methods: A Deeper Look

Overloading

```
1  // Fig. 6.10: MethodOverload.java
2  // Overloaded method declarations.
3
4  public class MethodOverload
5  {
6     // test overloaded square methods
7     public static void main(String[] args)
8     {
9        System.out.printf("Square of integer 7 is %d%n", square(7));
10       System.out.printf("Square of double 7.5 is %f%n", square(7.5));
11    }
12
13    // square method with int argument
14    public static int square(int intValue)
15    {
16       System.out.printf("%nCalled square with int argument: %d%n",
17          intValue);
18       return intValue * intValue;
19    }
20
21    // square method with double argument
22    public static double square(double doubleValue)
23    {
24       System.out.printf("%nCalled square with double argument: %f%n",
25          doubleValue);
26       return doubleValue * doubleValue;
27    }
28 } // end class MethodOverload
```

Exercise

(Temperature Conversions) Implement the following integer methods:

a) Method celsius returns the Celsius equivalent of a Fahrenheit temperature, using the calculation : celsius = 5.0 / 9.0 * (fahrenheit - 32);

b) Method fahrenheit returns the Fahrenheit equivalent of a Celsius temperature, using the calculation: fahrenheit = 9.0 / 5.0 * celsius + 32;

c) Use the methods from parts (a) and (b) to write an application that enables the user either to enter a Fahrenheit temperature and display the Celsius equivalent or to enter a Celsius temperature and display the Fahrenheit equivalent.

Arrays and ArrayLists

Arrays and ArrayLists

Creating and Initializing an Array

```java
// Fig. 7.2: InitArray.java
// Initializing the elements of an array to default values of zero.

public class InitArray
{
   public static void main(String[] args)
   {
      // declare variable array and initialize it with an array object
      int[] array = new int[10]; // create the array object

      System.out.printf("%s%8s%n", "Index", "Value"); // column headings

      // output each array element's value
      for (int counter = 0; counter < array.length; counter++)
         System.out.printf("%5d%8d%n", counter, array[counter]);
   }
} // end class InitArray
```

```java
// Fig. 7.3: InitArray.java
// Initializing the elements of an array with an array initializer.

public class InitArray
{
   public static void main(String[] args)
   {
      // initializer list specifies the initial value for each element
      int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

      System.out.printf("%s%8s%n", "Index", "Value"); // column headings

      // output each array element's value
      for (int counter = 0; counter < array.length; counter++)
         System.out.printf("%5d%8d%n", counter, array[counter]);

} // end class InitArray
```

Arrays and ArrayLists

Enhanced for Statement

```
for (parameter : arrayName)
   statement
```

```
1  // Fig. 7.12: EnhancedForTest.java
2  // Using the enhanced for statement to total integers in an array.
3
4  public class EnhancedForTest
5  {
6     public static void main(String[] args)
7     {
8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // add each element's value to total
12       for (int number : array)
13          total += number;
14
15       System.out.printf("Total of array elements: %d%n", total);
16    }
17 } // end class EnhancedForTest
```

Avoid the possibility of “stepping outside” the array

Arrays and ArrayLists

Passing Arrays to Methods

```
double[] hourlyTemperatures = new double[24];
```

```
void modifyArray(double[] b)
```

```
modifyArray(hourlyTemperatures);
```

```
1  // Fig. 7.13: PassArray.java
2  // Passing arrays and individual array elements to methods.
3
4  public class PassArray
5  {
6     // main creates array and calls modifyArray and modifyElement
7     public static void main(String[] args)
8     {
9        int[] array = { 1, 2, 3, 4, 5 };
10
11       System.out.printf(
12          "Effects of passing reference to entire array:%n" +
13          "The values of the original array are:%n");
14
15       // output original array elements
16       for (int value : array)
17          System.out.printf("   %d", value);
18
19       modifyArray(array); // pass array reference
20       System.out.printf("%n%nThe values of the modified array are:%n");
21
22       // output modified array elements
23       for (int value : array)
24          System.out.printf("   %d", value);
25
26       System.out.printf(
27          "%n%nEffects of passing array element value:%n" +
28          "array[3] before modifyElement: %d%n", array[3]);
29
30       modifyElement(array[3]); // attempt to modify array[3]
31       System.out.printf(
32          "array[3] after modifyElement: %d%n", array[3]);
33    }
34
35    // multiply each element of an array by 2
36    public static void modifyArray(int[] array2)
37    {
38       for (int counter = 0; counter < array2.length; counter++)
39          array2[counter] *= 2;
40    }
41
42    // multiply argument by 2
43    public static void modifyElement(int element)
44    {
45       element *= 2;
46       System.out.printf(
47          "Value of element in modifyElement: %d%n", element);
48    }
49 } // end class PassArray
```

Arrays and ArrayLists

Pass-by-Value vs. Pass-by-Reference

- **Unlike some other languages, Java does not allow you to choose pass-by-value or pass-by-reference—all arguments are passed by value;**
- **Objects themselves cannot be passed to methods;**
- **For example, the array[3] to method modifyElement displayed on the previous slide;**
- **This is apply for reference-type parameters too.**
- **makes sense for performance reasons.**

Arrays and ArrayLists

Two-Dimensional Array

```
1  // Fig. 7.17: InitArray.java
2  // Initializing two-dimensional arrays.
3
4  public class InitArray
5  {
6     // create and output two-dimensional arrays
7     public static void main(String[] args)
8     {
9        int[][] array1 = {{1, 2, 3}, {4, 5, 6}};
10       int[][] array2 = {{1, 2}, {3}, {4, 5, 6}};
11
12       System.out.println("Values in array1 by row are");
13       outputArray(array1); // displays array1 by row
14
15       System.out.printf("%nValues in array2 by row are%n");
16       outputArray(array2); // displays array2 by row
17    }
18
19    // output rows and columns of a two-dimensional array
20    public static void outputArray(int[][] array)
21    {
22       // loop through array's rows
23       for (int row = 0; row < array.length; row++)
24       {
25          // loop through columns of current row
26          for (int column = 0; column < array[row].length; column++)
27             System.out.printf("%d  ", array[row][column]);
28
29          System.out.println();
30       }
31    }
32 } // end class InitArray
```

Arrays and ArrayLists

Variable-Length Argument Lists

```
1  // Fig. 7.20: VarargsTest.java
2  // Using variable-length argument lists.
3
4  public class VarargsTest
5  {
6     // calculate average
7     public static double average(double... numbers)
8     {
9        double total = 0.0;
10
11       // calculate total using the enhanced for statement
12       for (double d : numbers)
13          total += d;
14
15       return total / numbers.length;
16    }
17
18    public static void main(String[] args)
19    {
20       double d1 = 10.0;
21       double d2 = 20.0;
22       double d3 = 30.0;
23       double d4 = 40.0;
24
25       System.out.printf("d1 = %.1f%nd2 = %.1f%nd3 = %.1f%nd4 = %.1f%n%n",
26          d1, d2, d3, d4);
27
28       System.out.printf("Average of d1 and d2 is %.1f%n",
29          average(d1, d2) );
30       System.out.printf("Average of d1, d2 and d3 is %.1f%n",
31          average(d1, d2, d3) );
32       System.out.printf("Average of d1, d2, d3 and d4 is %.1f%n",
33          average(d1, d2, d3, d4) );
34    }
35 } // end class VarargsTest
```

Receive an unspecified number of arguments.

Arrays and ArrayLists

Class Arrays

```java
// Fig. 7.20: VarargsTest.java
// Using variable-length argument lists.

public class VarargsTest
{
   // calculate average
   public static double average(double... numbers)
   {
      double total = 0.0;

      // calculate total using the enhanced for statement
      for (double d : numbers)
         total += d;

      return total / numbers.length;
   }

   public static void main(String[] args)
   {
      double d1 = 10.0;
      double d2 = 20.0;
      double d3 = 30.0;
      double d4 = 40.0;

      System.out.printf("d1 = %.1f%nd2 = %.1f%nd3 = %.1f%nd4 = %.1f%n%n",
         d1, d2, d3, d4);

      System.out.printf("Average of d1 and d2 is %.1f%n",
         average(d1, d2) );
      System.out.printf("Average of d1, d2 and d3 is %.1f%n",
         average(d1, d2, d3) );
      System.out.printf("Average of d1, d2, d3 and d4 is %.1f%n",
         average(d1, d2, d3, d4) );
   }
} // end class VarargsTest
```

```java
// Fig. 7.22: ArrayManipulations.java
// Arrays class methods and System.arraycopy.
import java.util.Arrays;

public class ArrayManipulations
{
   public static void main(String[] args)
   {
      // sort doubleArray into ascending order
      double[] doubleArray = { 8.4, 9.3, 0.2, 7.9, 3.4 };
      Arrays.sort(doubleArray);
      System.out.printf("%ndoubleArray: ");

      for (double value : doubleArray)
         System.out.printf("%.1f ", value);

      // fill 10-element array with 7s
      int[] filledIntArray = new int[10];
      Arrays.fill(filledIntArray, 7);
      displayArray(filledIntArray, "filledIntArray");

      // copy array intArray into array intArrayCopy
      int[] intArray = { 1, 2, 3, 4, 5, 6 };
      int[] intArrayCopy = new int[intArray.length];
      System.arraycopy(intArray, 0, intArrayCopy, 0, intArray.length);
      displayArray(intArray, "intArray");
      displayArray(intArrayCopy, "intArrayCopy");

      // compare intArray and intArrayCopy for equality
      boolean b = Arrays.equals(intArray, intArrayCopy);
      System.out.printf("%n%nintArray %s intArrayCopy%n",
         (b ? "==" : "!="));
```

Arrays and ArrayLists

Variable-Length Argument Lists

```
1  // Fig. 7.20: VarargsTest.java
2  // Using variable-length argument lists.
3
4  public class VarargsTest
5  {
6     // calculate average
7     public static double average(double... numbers)
8     {
9        double total = 0.0;
10
11       // calculate total using the enhanced for statement
12       for (double d : numbers)
13          total += d;
14
15       return total / numbers.length;
16    }
17
18    public static void main(String[] args)
19    {
20       double d1 = 10.0;
21       double d2 = 20.0;
22       double d3 = 30.0;
23       double d4 = 40.0;
24
25       System.out.printf("d1 = %.1f%nd2 = %.1f%nd3 = %.1f%nd4 = %.1f%n%n",
26          d1, d2, d3, d4);
27
28       System.out.printf("Average of d1 and d2 is %.1f%n",
29          average(d1, d2) );
30       System.out.printf("Average of d1, d2 and d3 is %.1f%n",
31          average(d1, d2, d3) );
32       System.out.printf("Average of d1, d2, d3 and d4 is %.1f%n",
33          average(d1, d2, d3, d4) );
34    }
35 } // end class VarargsTest
```

Receive an unspecified number of arguments.

Arrays and ArrayLists

ArrayList

```
ArrayList<String> list;
```

```
ArrayList<Integer> integers;
```

ArrayList<T> (package java.util)

Method	Description
add	Adds an element to the *end* of the ArrayList.
clear	Removes all the elements from the ArrayList.
contains	Returns true if the ArrayList contains the specified element; otherwise, returns false.
get	Returns the element at the specified index.
indexOf	Returns the index of the first occurrence of the specified element in the ArrayList.
remove	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
size	Returns the number of elements stored in the ArrayList.
trimToSize	Trims the capacity of the ArrayList to the current number of elements.

- **These classes provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored;**

- **This provides a convenient solution to this problem—it can dynamically change its size to accommodate more elements.**

Arrays and ArrayLists

ArrayList

```
1  // Fig. 7.24: ArrayListCollection.java
2  // Generic ArrayList<T> collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection
6  {
7     public static void main(String[] args)
8     {
9        // create a new ArrayList of Strings with an initial capacity of 10
10       ArrayList<String> items = new ArrayList<String>();
11
12       items.add("red"); // append an item to the list
13       items.add(0, "yellow"); // insert "yellow" at index 0
14
15       // header
16       System.out.print(
17          "Display list contents with counter-controlled loop:");
18
19       // display the colors in the list
20       for (int i = 0; i < items.size(); i++)
21          System.out.printf(" %s", items.get(i));
22
23       // display colors using enhanced for in the display method
24       display(items,
25          "%nDisplay list contents with enhanced for statement:");
26
27       items.add("green"); // add "green" to the end of the list
28       items.add("yellow"); // add "yellow" to the end of the list
29       display(items, "List with two new elements:");
30
31       items.remove("yellow"); // remove the first "yellow"
32       display(items, "Remove first instance of yellow:");
33
34       items.remove(1); // remove item at index 1
35       display(items, "Remove second list element (green):");
36
37       // check if a value is in the List
38       System.out.printf("\"red\" is %sin the list%n",
39          items.contains("red") ? "": "not ");
40
41       // display number of elements in the List
42       System.out.printf("Size: %s%n", items.size());
43    }
44
45    // display the ArrayList's elements on the console
46    public static void display(ArrayList<String> items, String header)
47    {
48       System.out.printf(header); // display header
49
50       // display each element in items
51       for (String item : items)
52          System.out.printf(" %s", item);
53
```

Classes and Objects

Time1 class

```
1  // Fig. 8.1: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3
4  public class Time1
5  {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // set a new time value using universal time; throw an
11    // exception if the hour, minute or second is invalid
12    public void setTime(int hour, int minute, int second)
13    {
14       // validate hour, minute and second
15       if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
16          second < 0 || second >= 60)
17       {
18          throw new IllegalArgumentException(
19             "hour, minute and/or second was out of range");
20       }
21
22       this.hour = hour;
23       this.minute = minute;
24       this.second = second;
25    }
26
27    // convert to String in universal-time format (HH:MM:SS)
28    public String toUniversalString()
29    {
30       return String.format("%02d:%02d:%02d", hour, minute, second);
31    }
32
33    // convert to String in standard-time format (H:MM:SS AM or PM)
34    public String toString()
35    {
36       return String.format("%d:%02d:%02d %s",
37          ((hour == 0 || hour == 12) ? 12 : hour % 12),
38          minute, second, (hour < 12 ? "AM" : "PM"));
39    }
40 } // end class Time1
```

Classes and Objects

Time1Test class

```
1  // Fig. 8.2: Time1Test.java
2  // Time1 object used in an app.
3
4  public class Time1Test
5  {
6     public static void main(String[] args)
7     {
8        // create and initialize a Time1 object
9        Time1 time = new Time1(); // invokes Time1 constructor
10
11       // output string representations of the time
12       displayTime("After time object is created", time);
13       System.out.println();
14
15       // change time and output updated time
16       time.setTime(13, 27, 6);
17       displayTime("After calling setTime", time);
18       System.out.println();
19
20       // attempt to set time with invalid values
21       try
22       {
23          time.setTime(99, 99, 99); // all values out of range
24       }
25       catch (IllegalArgumentException e)
26       {
27          System.out.printf("Exception: %s%n%n", e.getMessage());
28       }
29
30       // display time after attempt to set invalid values
31       displayTime("After calling setTime with invalid values", time);
32    }
33
34    // displays a Time1 object in 24-hour and 12-hour formats
35    private static void displayTime(String header, Time1 t)
36    {
37       System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
38          header, t.toUniversalString(), t.toString());
39    }
40 } // end class Time1Test
```

Classes and Objects

Controlling Access to Members

```
1  // Fig. 8.3: MemberAccessTest.java
2  // Private members of class Time1 are not accessible.
3  public class MemberAccessTest
4  {
5     public static void main(String[] args)
6     {
7        Time1 time = new Time1(); // create and initialize Time1 object
8
9        time.hour = 7; // error: hour has private access in Time1
10       time.minute = 15; // error: minute has private access in Time1
11       time.second = 30; // error: second has private access in Time1
12    }
13 } // end class MemberAccessTest
```

Classes and Objects

this

```
1  // Fig. 8.4: ThisTest.java
2  // this used implicitly and explicitly to refer to members of an object.
3
4  public class ThisTest
5  {
6     public static void main(String[] args)
7     {
8        SimpleTime time = new SimpleTime(15, 30, 19);
9        System.out.println(time.buildString());
10    }
11 } // end class ThisTest
12
13 // class SimpleTime demonstrates the "this" reference
14 class SimpleTime
15 {
16    private int hour; // 0-23
17    private int minute; // 0-59
18    private int second; // 0-59
19
20    // if the constructor uses parameter names identical to
21    // instance variable names the "this" reference is
22    // required to distinguish between the names
23    public SimpleTime(int hour, int minute, int second)
24    {
25       this.hour = hour; // set "this" object's hour
26       this.minute = minute; // set "this" object's minute
27       this.second = second; // set "this" object's second
28    }
29
30    // use explicit and implicit "this" to call toUniversalString
31    public String buildString()
32    {
33       return String.format("%24s: %s%n%24s: %s",
34          "this.toUniversalString()", this.toUniversalString(),
35          "toUniversalString()", toUniversalString());
36    }
37
38    // convert to String in universal-time format (HH:MM:SS)
39    public String toUniversalString()
40    {
41       // "this" is not required here to access instance variables,
42       // because method does not have local variables with same
43       // names as instance variables
44       return String.format("%02d:%02d:%02d",
45          this.hour, this.minute, this.second);
46    }
47 } // end class SimpleTime
```

Every object can access a reference to itself with keyword this (sometimes called the this reference).

Classes and Objects

Composition

```java
// Fig. 8.7: Date.java
// Date class declaration.

public class Date
{
   private int month; // 1-12
   private int day; // 1-31 based on month
   private int year; // any year

   private static final int[] daysPerMonth =
      { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   // constructor: confirm proper value for month and day given the year
   public Date(int month, int day, int year)
   {
      // check if month in range
      if (month <= 0 || month > 12)
         throw new IllegalArgumentException(
            "month (" + month + ") must be 1-12");

      // check if day in range for month
      if (day <= 0 ||
         (day > daysPerMonth[month] && !(month == 2 && day == 29)))
         throw new IllegalArgumentException("day (" + day +
            ") out-of-range for the specified month and year");

      // check for leap year if month is 2 and day is 29
      if (month == 2 && day == 29 && !(year % 400 == 0 ||
           (year % 4 == 0 && year % 100 != 0)))
         throw new IllegalArgumentException("day (" + day +
            ") out-of-range for the specified month and year");

      this.month = month;
      this.day = day;
      this.year = year;

      System.out.printf(
         "Date object constructor for date %s%n", this);
   }

   // return a String of the form month/day/year
   public String toString()
   {
      return String.format("%d/%d/%d", month, day, year);
   }
} // end class Date
```

A class can have references to objects of other classes as members. This is called composition and is sometimes referred to as a has-a relationship

Classes and Objects

Composition

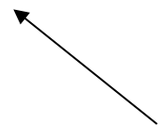
composition

```
1  // Fig. 8.8: Employee.java
2  // Employee class with references to other objects.
3
4  public class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
10
11    // constructor to initialize name, birth date and hire date
12    public Employee(String firstName, String lastName, Date birthDate,
13       Date hireDate)
14    {
15       this.firstName = firstName;
16       this.lastName = lastName;
17       this.birthDate = birthDate;
18       this.hireDate = hireDate;
19    }
20
21    // convert Employee to String format
22    public String toString()
23    {
24       return String.format("%s, %s  Hired: %s  Birthday: %s",
25          lastName, firstName, hireDate, birthDate);
26    }
27 } // end class Employee
```

Classes and Objects

Composition

```
1  // Fig. 8.9: EmployeeTest.java
2  // Composition demonstration.
3
4  public class EmployeeTest
5  {
6     public static void main(String[] args)
7     {
8        Date birth = new Date(7, 24, 1949);
9        Date hire = new Date(3, 12, 1988);
10       Employee employee = new Employee("Bob", "Blue", birth, hire);
11
12       System.out.println(employee);
13    }
14 } // end class EmployeeTest
```



Implicitly invokes the Employee ’s toString

Classes and Objects

static Class Members

```
1  // Fig. 8.12: Employee.java
2  // static variable used to maintain a count of the number of
3  // Employee objects in memory.
4
5  public class Employee
6  {
7     private static int count = 0; // number of Employees created
8     private String firstName;
9     private String lastName;
10
11    // initialize Employee, add 1 to static count and
12    // output String indicating that constructor was called
13    public Employee(String firstName, String lastName)
14    {
15       this.firstName = firstName;
16       this.lastName = lastName;
17
18       ++count;  // increment static count of employees
19       System.out.printf("Employee constructor: %s %s; count = %d%n",
20          firstName, lastName, count);
21    }
22
23    // get first name
24    public String getFirstName()
25    {
26       return firstName;
27    }
28
29    // get last name
30    public String getLastName()
31    {
32       return lastName;
33    }
34
35    // static method to get static count value
36    public static int getCount()
37    {
38       return count;
39    }
40 } // end class Employee
```

We can access a class's public static members in two ways:

- **Through a reference to any object of the class,**
- **Qualifying the member name with the class name and a dot (.), as in Math.random() .**

Classes and Objects

static Class Members

```
1  // Fig. 8.13: EmployeeTest.java
2  // static member demonstration.
3
4  public class EmployeeTest
5  {
6     public static void main(String[] args)
7     {
8        // show that count is 0 before creating Employees
9        System.out.printf("Employees before instantiation: %d%n",
10          Employee.getCount());
11
12       // create two Employees; count should be 2
13       Employee e1 = new Employee("Susan", "Baker");
14       Employee e2 = new Employee("Bob", "Blue");
15
16       // show that count is 2 after creating two Employees
17       System.out.printf("%nEmployees after instantiation:%n");
18       System.out.printf("via e1.getCount(): %d%n", e1.getCount());
19       System.out.printf("via e2.getCount(): %d%n", e2.getCount());
20       System.out.printf("via Employee.getCount(): %d%n",
21          Employee.getCount());
22
23       // get names of Employees
24       System.out.printf("%nEmployee 1: %s %s%nEmployee 2: %s %s%n",
25          e1.getFirstName(), e1.getLastName(),
26          e2.getFirstName(), e2.getLastName());
27    }
28 } // end class EmployeeTest
```

Static Method.

Classes and Objects

Package Access

```
1  // Fig. 8.15: PackageDataTest.java
2  // Package-access members of a class are accessible by other classes
3  // in the same package.
4
5  public class PackageDataTest
6  {
7     public static void main(String[] args)
8     {
9        PackageData packageData = new PackageData();
10
11       // output String representation of packageData
12       System.out.printf("After instantiation:%n%s%n", packageData);
13
14       // change package access data in packageData object
15       packageData.number = 77;
16       packageData.string = "Goodbye";
17
18       // output String representation of packageData
19       System.out.printf("%nAfter changing values:%n%s%n", packageData);
20    }
21 } // end class PackageDataTest
22
23 // class with package access instance variables
24 class PackageData
25 {
26    int number; // package-access instance variable
27    String string; // package-access instance variable
28
29    // constructor
30    public PackageData()
31    {
32       number = 0;
33       string = "Hello";
34    }
35
36    // return PackageData object String representation
37    public String toString()
38    {
39       return String.format("number: %d; string: %s", number, string);
40    }
41 } // end class PackageData
```

Projects