Chapter 5
Introduction to Defining Classes

Fundamentals of Java

Objectives

- Design and implement a simple class from user requirements.
- Organize a program in terms of a view class and a model class.
- Use visibility modifiers to make methods visible to clients and restrict access to data within a class.

Objectives (cont.)

- Write appropriate mutator methods, accessor methods, and constructors for a class.
- Understand how parameters transmit data to methods.
- Use instance variables, local variables, and parameters appropriately.
- Organize a complex task in terms of helper methods.

Vocabulary

- Accessor
- Actual parameter
- Behavior
- Constructor
- Encapsulation

Vocabulary (cont.)

- Formal parameter
- Helper method
- Identity
- Instantiation
- Lifetime

Vocabulary (cont.)

- Mutator
- Scope
- State
- Visibility modifier

The Internal Structure of Classes and Objects

- A class is a template that describes the characteristics of similar objects.
  - Variable declarations define an object's data.
    - **Instance variables**
  - Methods define an object's behavior in response to messages.

The Internal Structure of Classes and Objects (cont.)

- **Encapsulation:** Combining data and behavior into a single software package
- An object is an **instance** of its class.
- **Instantiation:** Process of creating a new object

The Internal Structure of Classes and Objects (cont.)

- During execution, a computer's memory holds:
  - All class templates in their compiled form
  - Variables that refer to objects
  - Objects as needed
- Memory for data is allocated within objects.
- Objects appear and occupy memory when instantiated.
  - Disappear when no longer needed

The Internal Structure of Classes and Objects (cont.)

- **Garbage collection:** JVM's automated method for removing unused objects
  - Tracks whether objects are referenced by any variables
- Three characteristics of an object:
  - Behavior (methods)
  - State (data values)
  - Identity (unique ID for each object)

The Internal Structure of Classes and Objects (cont.)

- When messages sent, two objects involved:
  - **Client:** The message sender
    - Only needs to know the **interface** of the server
  - **Server:** The message receiver
    - Supports and implements an interface
- **Information hiding:** Server's data requirements and method implementation hidden from client

A Student Class

METHODS	DESCRIPTIONS
`void setName(aString)`	Example: `stu.setName ("Bill");` sets the name of `stu` to Bill
`String getName()`	Example: `str = stu.getName();` returns the name of `stu`
`void setScore (whichTest, testScore)`	Example: `stu.setScore (3, 95);` sets the score on test 3 to 95 if `whichTest` is not 1, 2, or 3, then 3 is substituted automatically
`int getScore(whichTest)`	Example: `score = stu.getScore (3);` returns the score on test 3 if `whichTest` is not 1, 2, or 3, then 3 is substituted automatically
`int getAverage()`	Example : `average = stu.getAverage();` returns the average of the test scores
`int getHighScore()`	Example: `highScore = stu.getHighScore();` returns the highest test score
`String toString()`	Example: `str = stu.toString();` returns a string containing the student's name and test scores

Table 5-1: Interface for the `Student` class

A `Student` Class: Using `Student` Objects

- Declare and instantiate a `Student` object:
  - `Student s1 = new Student();`
- Sending messages to a `Student` object:
  - `String str = s1.getName();`
  - `s1.setName("Bill");`
  - `System.out.println(s1.toString());`

A `Student` Class: Objects, Assignment, and Aliasing

- Multiple variables can point at the same object
  - Example:
    - ```
      Student s1 = new Student();
      Student s2 = s1;
      ```
- To cause a variable to no longer point at any object, set it equal to `null`, as in:
  - `s1 = null;`
      ```

A Student Class: Objects, Assignment, and Aliasing (cont.)

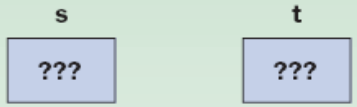
CODE	DIAGRAM	COMMENTS
`int i, j;`		`i` and `j` are memory locations that have not yet been initialized, but which will hold integers.
`i = 3;` `j = i;`		`i` holds the integer `3`. `j` holds the integer `3`.
`Student s, t;`		`s` and `t` are memory locations that have not yet been initialized, but which will hold references to `Student` objects.

Table 5-2: How variables are affected by assignment statements

A Student Class: Objects, Assignment, and Aliasing (cont.)

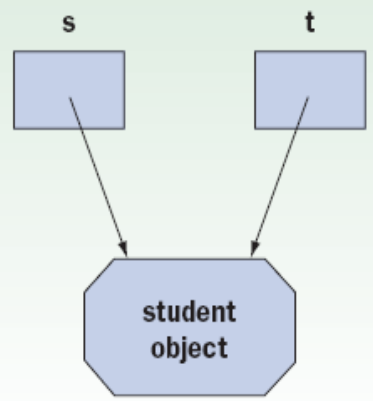
CODE	DIAGRAM	COMMENTS
`s = new Student();` `t = s;`		`s` holds a reference to a `Student` object. `t` holds a reference to the same `Student` object.

Table 5-2: How variables are affected by assignment statements (cont.)

A `Student` Class (cont.)

- Two fundamental data type categories:
  - **Primitive types:** `int`, `double`, `boolean`, `char`
    - Shorter and longer versions of these types
  - **Reference types:** All classes

```
int number = 45;
String word = "Hi";
```

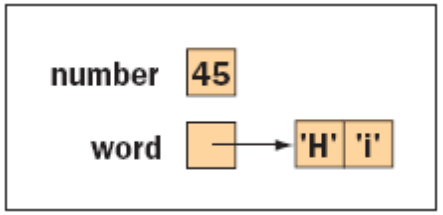


Figure 5-2: Difference between primitive and reference variables

A Student Class (cont.)

```
Student student = new Student("Mary", 70, 80, 90);
student = null;
```

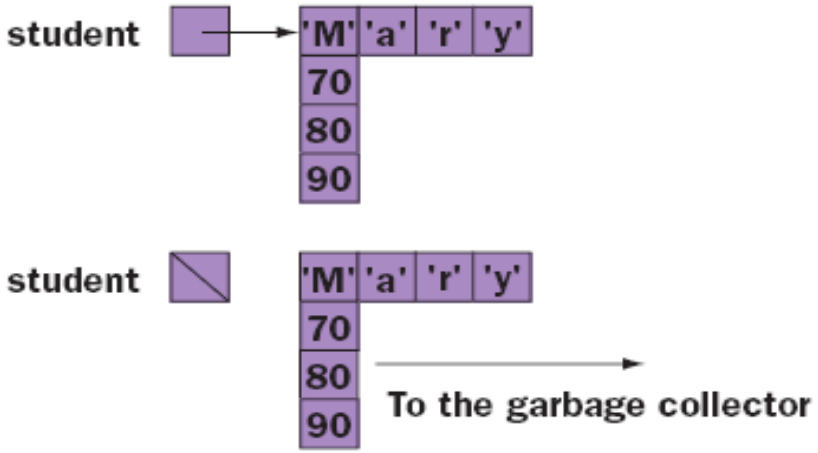


Figure 5-3: `Student` variable before and after it has been assigned the value `null`

A `Student` Class (cont.)

- Can compare a reference variable to `null`
  - Avoid **null pointer exceptions**

```
if (student == null)
   …                    // Don't try to run a method with that student!
else
   …                    // Process the student

while (student != null){
   …                    // Process the student
                        // Obtain the next student from whatever source
}
```

A Student Class: The Structure of a Class Template

```
public class <name of class> extends <some other class>{

    // Declaration of instance variables
    private <type> <name>;
    ...

    // Code for the constructor methods
    public <name of class>() {
    // Initialize the instance variables
       ...
    }
    ...

    // Code for the other methods
    public <return type> <name of method> (<parameter list>){
       ...
    }
    ...
}
```

A `Student` Class: The Structure of a Class Template (cont.)

- **`public`**: Class is accessible to anyone
- Name of class must follow Java naming conventions
- **`extends: Optional`**
  - Java organizes class in a hierarchy.
  - If Class B extends Class A, it inherits instance variables and methods from Class A.

A `Student` Class: The Structure of a Class Template (cont.)

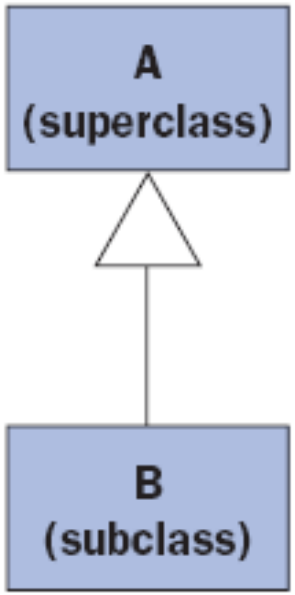


Figure 5.4: Relationship between superclass and subclass

A `Student` Class: The Structure of a Class Template (cont.)

- `private` and `public` are **visibility modifiers**.
  - Define whether a method or instance variable can be seen outside of the class
    - Instance variables should generally be private.

A `Student` Class: Constructors (cont.)

- Initialize a newly instantiated object's instance variables
  - Activated (called) only by the keyword `new`
- **Default constructors:** Empty parameter lists
- A class is easier to use when it has a variety of constructors.

A Student Class: Chaining Constructors (cont.)

```
// Default constructor -- initialize name to the empty string and
// the test scores to zero.
public Student(){
   this("", 0, 0, 0);
}

// Additional constructor -- initialize the name and test scores
// to the values provided.
public Student(String nm, int t1, int t2, int t3){
   name = nm;
   test1 = t1;
   test2 = t2;
   test3 = t3;
}

// Additional constructor -- initialize the name and test scores
// to match those in the parameter s.
public Student(Student s){
   this(s.name, s.test1, s.test2, s.test3);
}
```

Editing, Compiling, and Testing the `Student` Class

- Steps:
  - Save source code in *Student.java*.
  - Run `javac Student.java`.
  - Run/test the program.

Editing, Compiling, and Testing the `Student` Class (cont.)

```
public class TestStudent{

   public static void main (String[] args){
      Student s1, s2;

      s1 = new Student();     // Instantiate a student object
      s1.setName("Bill");     // Set the student's name to "Bill"
      s1.setScore(1,84);      // Set the score on test 1 to 84
      s1.setScore(2,86);      //                on test 2 to 86
      s1.setScore(3,88);      //                on test 3 to 88
      System.out.println("\nHere is student s1\n" + s1);

            s2 = s1;          // s1 and s2 now refer to the same object
      s2.setName("Ann");      // Set the name through s2
      System.out.println("\nName of s1 is now: " + s1.getName());
   }
}
```

Example 5.1: Tester program for the `Student` class

Editing, Compiling, and Testing the `Student` Class (cont.)

- Introduce an error into the `Student` class:

```
public int getAverage(){
    int average = 0;
    average = (int) Math.round((test1 + test2 + test3) / average);
    return average;
}
```

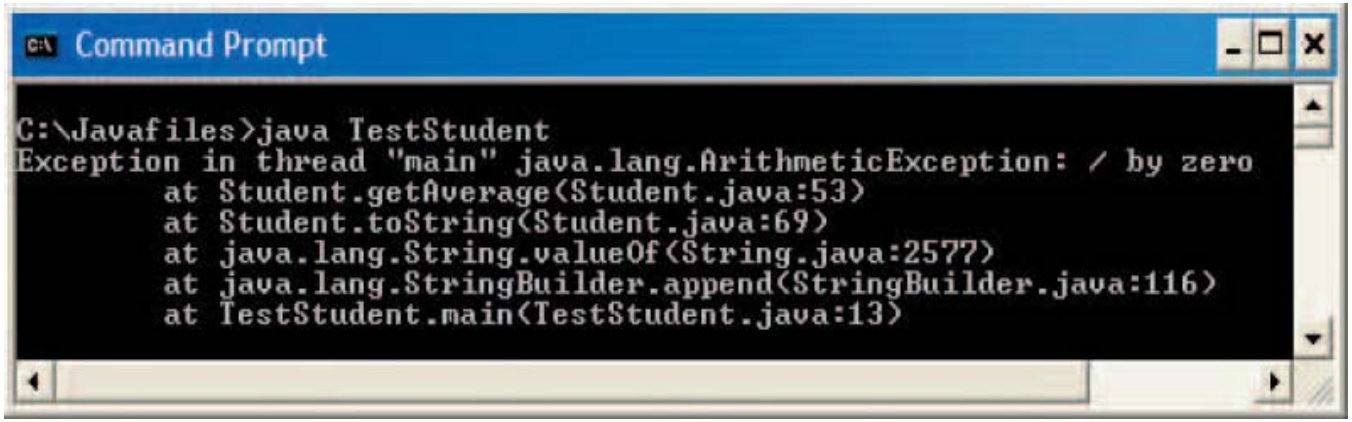


Figure 5-6: Divide by zero run-time error message

The Structure and Behavior of Methods

- Methods take the following form:

```
<visibility modifier> <return type> <method name> (<parameter list>){

   <implementing code>
}
```

- If the method returns no value, the return type should be `void`.

The Structure and Behavior of Methods (cont.)

- **`return` statements:** If a method has a return type, implementation must have at least one return statement that returns a value of that type.
  - A `return` statement in a `void` method simply ends the method.
  - Can have multiple `return` statements

The Structure and Behavior of Methods (cont.)

- **Formal parameters:** Parameters listed in a method's definition
- **Actual parameters (arguments):** Values passed to a method when it is invoked
- Parameter passing example:

```
// Client code

Student s = new Student();
Scanner reader = new Scanner(System.in);
System.out.print("Enter a test score:");
int testScore = reader.nextInt();
s.setScore(1, testScore);
```

```
// Server code

public void setScore (int i, int score){
   if      (i == 1) test1 = score;
   else if (i == 2) test2 = score;
   else             test3 = score;
}
```

The Structure and Behavior of Methods (cont.)

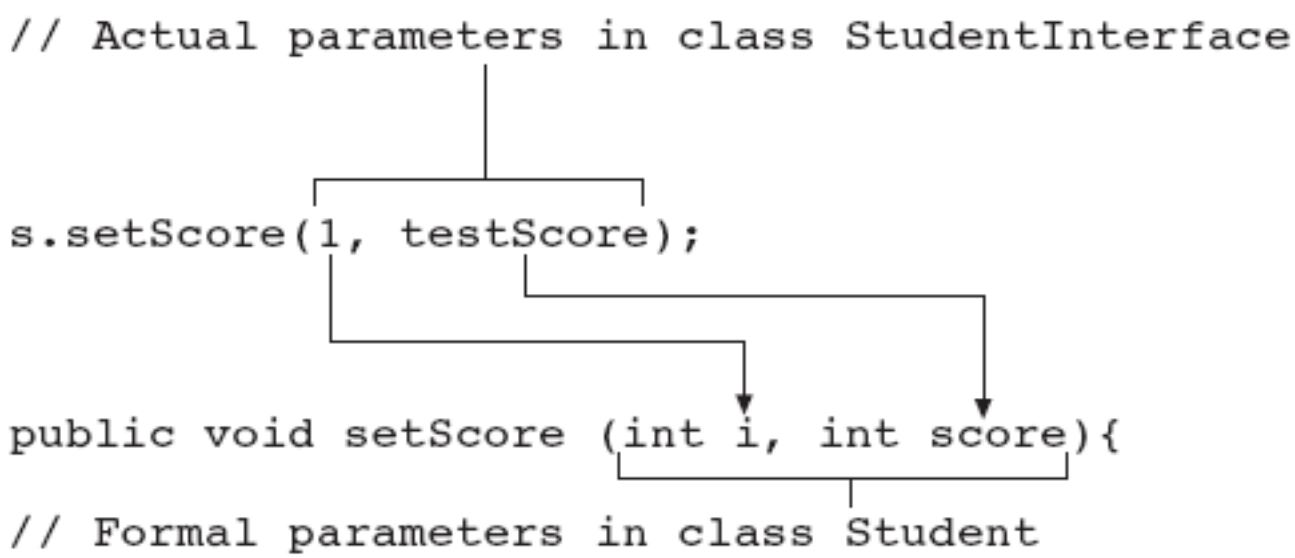


Figure 5.8: Parameter passing

The Structure and Behavior of Methods (cont.)

- **Helper methods:** Perform a piece of a task
  - Used by another method to perform a larger task
  - Usually private
    - Only methods already defined within the class need to use them
- When an object is instantiated, it receives own copy of its class's instance variables

Scope and Lifetime of Variables

- **Global variables:** Declared inside a class but outside any method
  - Accessible to any method in the class
- **Local variables:** Declared inside a method
  - Accessible only within that method

Scope and Lifetime of Variables (cont.)

- **Scope (of a variable):** Region where a variable can validly appear in lines of code
- Variables declared within any compound statement enclosed in braces have **block scope**.
  - Visible only within code enclosed by braces

Scope and Lifetime of Variables (cont.)

- **Lifetime:** Period when a variable can be used
  - Local variables exist while the method executes.
  - Instance variables exist while the object exists.
- Duplicate variable names may exist.
  - Local variables in different scopes
  - A local and a global variable
    - Local overrides global
    - Use `this` keyword to access global variable.

Scope and Lifetime of Variables (cont.)

```
public class ScopeDemo {

   private int iAmAVariable;

   public void someMethod (int parm){

      int iAmAVariable;
      ...
      iAmAVariable = 3;              // Refers to the local variable
      this.iAmAVariable = 4;         // Refers to the global variable
      ...
   }

   public void someOtherMethod(int iAmAVariable){
      ...
      this.iAmAVariable = iAmAVariable;     // Assign the value of the

  // parameter
            ...
  // to the global variable
               }
               ...
     }
```

Scope and Lifetime of Variables (cont.)

- Use instance variables to retain data.
  - Using local variables will result in lost data.
- Use local variables for temporary storage.
  - Using global variables could cause difficult-to-resolve logic errors.
- Use method parameters rather than global variables whenever possible.

Graphics and GUIs: Images

- To load an image:
  - `ImageIcon image = new ImageIcon(fileName);`
    - `fileName` indicates the location and name of a file containing an image.
- To paint an ImageIcon from a panel class:
  - `anImageIcon.paintIcon(this, g, x, y);`
  - `g` is the graphics context.
  - `x` and `y` are panel coordinates.

Graphics and GUIs: A `Circle` Class

- Useful to represent shapes as objects
  - A shape has attributes (color, size, position).
  - Can create more shapes than the `Graphics` class can draw
  - Given its graphics context, a shape can draw itself.
  - Easier to manipulate

Graphics and GUIs: A `Circle` Class (cont.)

METHOD	WHAT IT DOES
`Circle(int x, int y, int r, Color c)`	Constructor; creates a circle with center point (`x`, `y`), radius `r`, and color `c`
`int getX()`	Returns the `x` coordinate of the center
`int getY()`	Returns the `y` coordinate of the center
`int getRadius()`	Returns the radius
`Color getColor()`	Returns the color
`void setX(int x)`	Modifies the `x` coordinate of the center
`void setY(int y)`	Modifies the `y` coordinate of the center
`void setRadius(int r)`	Modifies the radius

Table 5-4: Methods in class `Circle`

Graphics and GUIs: A `Circle` Class (cont.)

METHOD	WHAT IT DOES
`Color setColor(Color c)`	Modifies the color
`void draw(Graphics g)`	Draws an outline of the circle in the graphics context
`void fill(Graphics g)`	Draws a filled circle in the graphics context
`boolean containsPoint(int x, int y)`	Returns true if the point (`x`, `y`) lies in the circle or false otherwise
`void move(int xAmount, int yAmount)`	Moves the circle by `xAmount` horizontally to the right and `yAmount` vertically downward; Negative amounts move to the left and up.

Table 5-4: Methods in class `Circle` (cont.)

Graphics and GUIs: A `Circle` Class (cont.)

```
import javax.swing.*;
import java.awt.*;

public class ColorPanel extends JPanel{

   private Circle c1, c2;

   public ColorPanel(Color backColor){
      setBackground(backColor);
      c1 = new Circle(200, 100, 25, Color.red);
      c2 = new Circle(100, 100, 50, Color.blue);
   }
   public void paintComponent(Graphics g){
      super.paintComponent(g);
      c1.fill(g);
      c2.draw(g);
   }
}
```

Example 5.3: Displays a circle and a filled circle

Graphics and GUIs: A `Circle` Class (cont.)

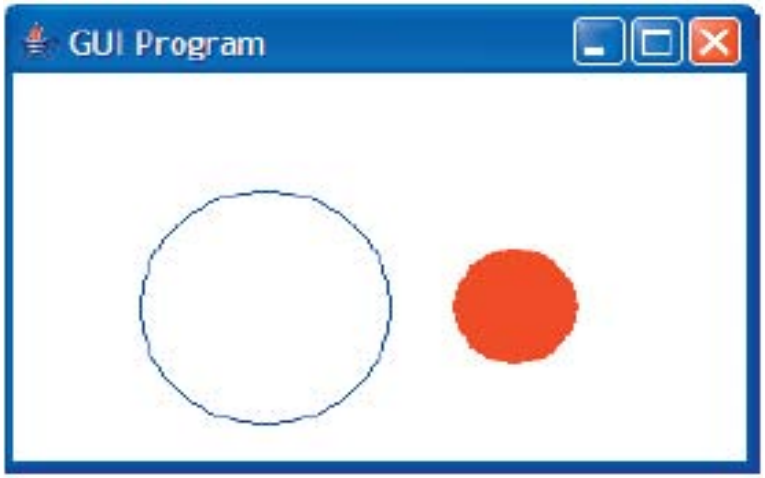


Figure 5-10: Displaying two `Circle` objects

Graphics and GUIs: A `Circle` Class (cont.)

- **`repaint` method:** Forces a refresh of any GUI component
  - Invokes object's `paintComponent` method
  - Called automatically by the JVM whenever the component is moved or altered
  - Can programatically call `repaint()` as well

Graphics and GUIs: Mouse Events

- Program can detect and respond to mouse events by attaching **listener objects** to a panel
  - When a particular type of mouse event occurs in a panel, its listeners are informed.
  - Listener class for capturing mouse click events extends `MouseAdapter`
  - Listener class for capturing mouse motion and dragging events extends `MouseMotionAdapter`

Graphics and GUIs: Mouse Events (cont.)

CLASS	METHOD	TYPE OF EVENT
MouseAdapter	public void mouseEntered(MouseEvent e) public void mouseExited(MouseEvent e) public void mousePressed(MouseEvent e) public void mouseReleased(MouseEvent e) public void mouseClicked(MouseEvent e)	Enter Exit Press Release Click
MouseMotionAdapter	public void mouseMoved(MouseEvent e) public void mouseDragged(MouseEvent e)	Move Drag

Table 5-5: Methods for responding to mouse events

Graphics and GUIs: Mouse Events (cont.)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;            //For the mouse events

public class ColorPanel extends JPanel{

   private Circle c1, c2;
   private Circle selectedCircle;   // Used to track selected shape
   private int x, y;                // Used to track mouse coordinates
```

Example 5.5: Displays a circle and a filled circle. Allows the user to drag a circle to another position

Graphics and GUIs: Mouse Events (cont.)

```
public ColorPanel(Color backColor){
    setBackground(backColor);
    c1 = new Circle(200, 100, 25, Color.red);
    c2 = new Circle(100, 100, 50, Color.blue);
    selectedCircle = null;
    addMouseListener(new PanelListener());
    addMouseMotionListener(new PanelMotionListener());
}

public void paintComponent(Graphics g){
    super.paintComponent(g);
    c1.fill(g);
    c2.draw(g);
}
```

Example 5.5: Displays a circle and a filled circle. Allows the user to drag a circle to another position (cont.)

Graphics and GUIs: Mouse Events (cont.)

```
private class PanelListener extends MouseAdapter{

   public void mousePressed(MouseEvent e){
      // Select a circle if it contains the mouse coordinates
      x = e.getX();
      y = e.getY();
      if (c1.containsPoint(x, y))
         selectedCircle = c1;
      else if (c2.containsPoint(x, y))
         selectedCircle = c2;
   }

   public void mouseReleased(MouseEvent e){
      // Deselect the selected circle
      x = e.getX();
      y = e.getY();
      selectedCircle = null;
   }
}
```

Example 5.5: Displays a circle and a filled circle. Allows the user to drag a circle to another position (cont.)

Graphics and GUIs: Mouse Events (cont.)

```
    private class PanelMotionListener extends MouseMotionAdapter{

        public void mouseDragged(MouseEvent e){
            // Compute the distance and move the selected circle
            int newX = e.getX();
            int newY = e.getY();
            int dx = newX - x;
            int dy = newY - y;
            if (selectedCircle != null)
                selectedCircle.move(dx, dy);
            x = newX;
            y = newY;
            repaint();
        }
    }
}
```

Example 5.5: Displays a circle and a filled circle. Allows the user to drag a circle to another position (cont.)

Summary

- Java class definitions consist of instance variables, constructors, and methods.
- Constructors initialize an object's instance variables when the object is created.
- A default constructor expects no parameters and sets the variables to default values.
- Mutator methods modify an object's instance variables.

Summary (cont.)

- Accessor methods allow clients to observe the values of these variables.
- The visibility modifier `public` makes methods visible to clients.
- `private` encapsulates access.
- Helper methods are called from other methods in a class definition.
  - Usually declared to be private

Summary (cont.)

- Instance variables track the state of an object.
- Local variables are used for temporary working storage within a method.
- Parameters transmit data to a method.
- A formal parameter appears in a method's signature and is referenced in its code.

Summary (cont.)

- Actual parameter is a value passed to a method when it is called.
- Scope of an instance variable is the entire class within which it is declared.
- Scope of a local variable or a parameter is the body of the method where it is declared.

Summary (cont.)

- Lifetime of an instance variable is the same as the lifetime of a particular object.
- Lifetime of a local variable and a parameter is the time during which a particular call of a method is active.