Chapter 9
Inheritance

Java Software Solutions

Foundations of Program Design

8th Edition

John Lewis
William Loftus

Inheritance

- Inheritance is a fundamental object-oriented design technique used to create and organize reusable classes

- Chapter 9 focuses on:

  - deriving new classes from existing classes
  - the `protected` modifier
  - creating class hierarchies
  - abstract classes
  - indirect visibility of inherited members
  - designing for inheritance
  - the GUI component class hierarchy
  - extending listener adapter classes
  - the `Timer` class

Outline

→ **Creating Subclasses**

Overriding Methods

Class Hierarchies

Visibility

**Designing for Inheritance**

Inheritance and GUIs

The `Timer` Class

Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one

- The existing class is called the *parent class,* or *superclass*, or *base class*

- The derived class is called the *child class* or *subclass*

- As the name implies, the child inherits characteristics of the parent

- That is, the child class inherits the methods and data defined by the parent class

Inheritance

- Inheritance relationships are shown in a UML class diagram using a solid arrow with an unfilled triangular arrowhead pointing to the parent class
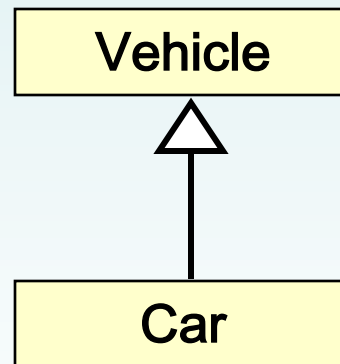
- Proper inheritance creates an *is-a* relationship, meaning the child *is a* more specific version of the parent

Inheritance

- A programmer can tailor a derived class as needed by adding new variables or methods, or by modifying the inherited ones

- One benefit of inheritance is *software reuse*

- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```java
public class Car extends Vehicle
{
    // class contents
}
```

- See `Words.java`
- See `Book.java`
- See `Dictionary.java`

```java
//***********************************************************************
//   Words.java          Author: Lewis/Loftus
//
//   Demonstrates the use of an inherited method.
//***********************************************************************

public class Words
{
    //--------------------------------------------------------------
    //   Instantiates a derived class and invokes its inherited and
    //   local methods.
    //--------------------------------------------------------------
    public static void main(String[] args)
    {
        Dictionary webster = new Dictionary();

        System.out.println("Number of pages: " + webster.getPages());

        System.out.println("Number of definitions: " +
                            webster.getDefinitions());

        System.out.println("Definitions per page: " +
                            webster.computeRatio());
    }
}
```

```java
//****************************     *****************************
//   Words.java
//
//   Demonstrates
//****************************     *****************************

public class Words
{
    //------------------------------------------------------------
    //   Instantiates a derived class and invokes its inherited and
    //   local methods.
    //------------------------------------------------------------
    public static void main(String[] args)
    {
        Dictionary webster = new Dictionary();

        System.out.println("Number of pages: " + webster.getPages());

        System.out.println("Number of definitions: " +
                            webster.getDefinitions());

        System.out.println("Definitions per page: " +
                            webster.computeRatio());
    }
}
```

Output

```
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35.0
```

```java
//***********************************************************
//  Book.java        Author: Lewis/Loftus
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance.
//***********************************************************

public class Book
{
   protected int pages = 1500;

   //----------------------------------------------------------
   //  Pages mutator.
   //----------------------------------------------------------
   public void setPages(int numPages)
   {
      pages = numPages;
   }


   //----------------------------------------------------------
   //  Pages accessor.
   //----------------------------------------------------------
   public int getPages()
   {
      return pages;
   }
}
```

```java
//***********************************************************
//  Dictionary.java        Author: Lewis/Loftus
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  inheritance.
//***********************************************************

public class Dictionary extends Book
{
   private int definitions = 52500;

   //-----------------------------------------------------------
   //  Prints a message using both local and inherited values.
   //-----------------------------------------------------------
   public double computeRatio()
   {
      return (double) definitions/pages;
   }

continue
```

continue

```java
    //-------------------------------------------------------
    //  Definitions mutator.
    //-------------------------------------------------------
    public void setDefinitions(int numDefinitions)
    {
        definitions = numDefinitions;
    }


    //-------------------------------------------------------
    //  Definitions accessor.
    //-------------------------------------------------------
    public int getDefinitions()
    {
        return definitions;
    }
}
```
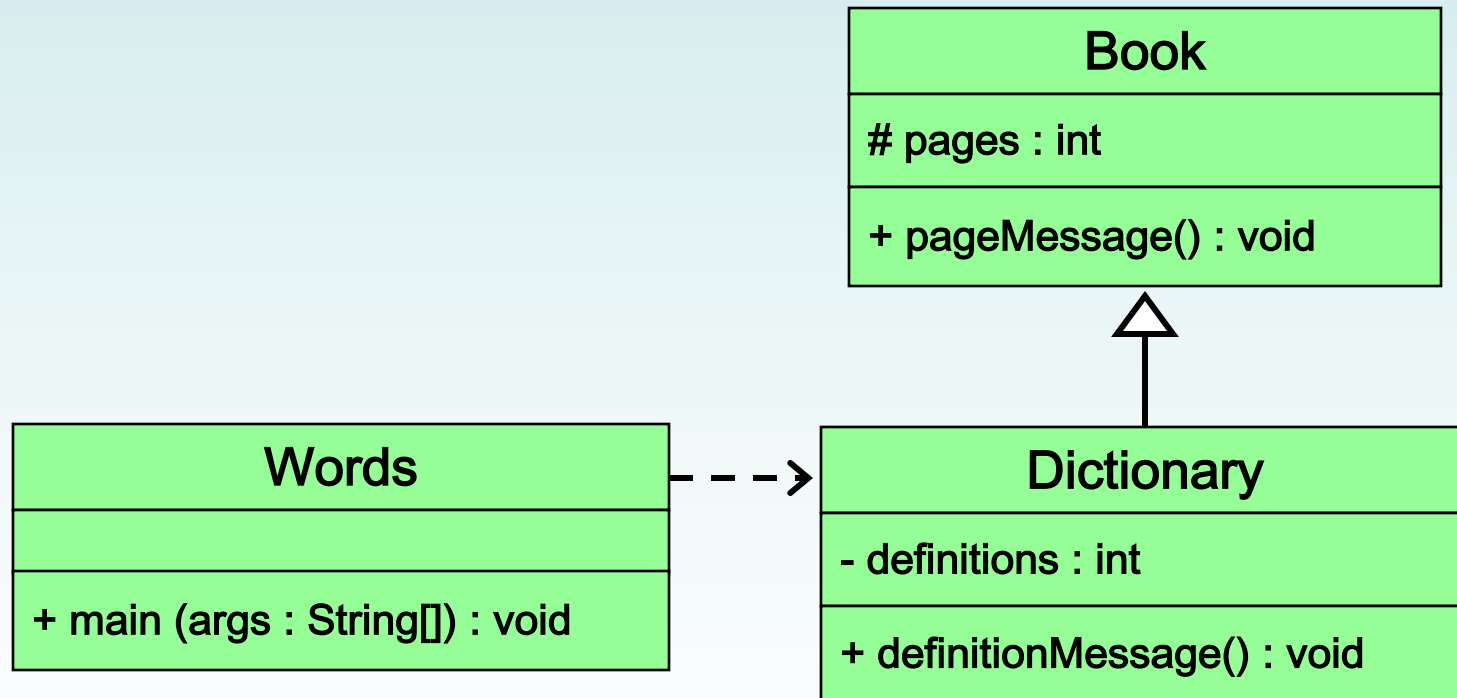
The protected Modifier

- Visibility modifiers affect the way that class members can be used in a child class

- Variables and methods declared with private visibility cannot be referenced in a child class

- They can be referenced in the child class if they are declared with public visibility -- but public variables violate the principle of encapsulation

- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a child class to reference a variable or method in the child class

- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility

- A protected variable is also visible to any class in the same package as the parent class

- See Appendix E for details of all Java modifiers

- Protected variables and methods can be shown with a # symbol preceding them in UML diagrams

Class Diagram for Words

Book

\# pages : int

\+ pageMessage() : void

Words

\+ main (args : String[]) : void

Dictionary

\- definitions : int

\+ definitionMessage() : void

The super Reference

- Constructors are not inherited, even though they have public visibility

- Yet we often want to use the parent's constructor to set up the "parent's part" of the object

- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

- A child's constructor is responsible for calling the parent's constructor

The super Reference

- The first line of a child's constructor should use the `super` reference to call the parent's constructor

- The `super` reference can also be used to reference other variables and methods defined in the parent's class

- See `Words2.java`
- See `Book2.java`
- See `Dictionary2.java`

```java
//*****************************************************************
//   Words2.java         Author: Lewis/Loftus
//
//   Demonstrates the use of the super reference.
//*****************************************************************

public class Words2
{
    //---------------------------------------------------------
    //   Instantiates a derived class and invokes its inherited and
    //   local methods.
    //---------------------------------------------------------
    public static void main(String[] args)
    {
        Dictionary2 webster = new Dictionary2(1500, 52500);

        System.out.println("Number of pages: " + webster.getPages());

        System.out.println("Number of definitions: " +
                            webster.getDefinitions());

        System.out.println("Definitions per page: " +
                            webster.computeRatio());
    }
}
```

```java
//*****************           *****************
//  Words2.java
//                   Output
//  Demonstrates     Number of pages: 1500
//*****************  Number of definitions: 52500   *****************
                    Definitions per page: 35.0

public class Words2
{
   //-----------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------
   public static void main(String[] args)
   {
      Dictionary2 webster = new Dictionary2(1500, 52500);

      System.out.println("Number of pages: " + webster.getPages());

      System.out.println("Number of definitions: " +
                         webster.getDefinitions());

      System.out.println("Definitions per page: " +
                         webster.computeRatio());
   }
}
```

```java
//***************************************************************
//   Book2.java         Author: Lewis/Loftus
//
//   Represents a book. Used as the parent of a derived class to
//   demonstrate inheritance and the use of the super reference.
//***************************************************************

public class Book2
{
    protected int pages;

    //------------------------------------------------------------
    //   Constructor: Sets up the book with the specified number of
    //   pages.
    //------------------------------------------------------------
    public Book2(int numPages)
    {
        pages = numPages;
    }

continue
```

continue

```
//-------------------------------------------------------
//  Pages mutator.
//-------------------------------------------------------
public void setPages(int numPages)
{
    pages = numPages;
}


//-------------------------------------------------------
//  Pages accessor.
//-------------------------------------------------------
public int getPages()
{
    return pages;
}
}
```

```java
//************************************************************
//   Dictionary2.java        Author: Lewis/Loftus
//
//   Represents a dictionary, which is a book. Used to demonstrate
//   the use of the super reference.
//************************************************************

public class Dictionary2 extends Book2
{
    private int definitions;

    //----------------------------------------------------------
    //   Constructor: Sets up the dictionary with the specified number
    //   of pages and definitions.
    //----------------------------------------------------------
    public Dictionary2(int numPages, int numDefinitions)
    {
        super(numPages);

        definitions = numDefinitions;
    }

continue
```

continue

```java
    //----------------------------------------------------------
    //   Prints a message using both local and inherited values.
    //----------------------------------------------------------
    public double computeRatio()
    {
       return (double) definitions/pages;
    }

    //----------------------------------------------------------
    //   Definitions mutator.
    //----------------------------------------------------------
    public void setDefinitions(int numDefinitions)
    {
       definitions = numDefinitions;
    }

    //----------------------------------------------------------
    //   Definitions accessor.
    //----------------------------------------------------------
    public int getDefinitions()
    {
       return definitions;
    }
}
```

Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class

- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents

- Collisions, such as the same variable name in two parents, have to be resolved

- Multiple inheritance is generally not needed, and Java does not support it

Outline

Creating Subclasses

→ **Overriding Methods**

Class Hierarchies

Visibility

Designing for Inheritance

Inheritance and GUIs

The `Timer` Class

Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own

- The new method must have the same signature as the parent's method, but can have a different body

- The type of the object executing the method determines which version of the method is invoked

- See `Messages.java`
- See `Thought.java`
- See `Advice.java`

```java
//*************************************************************
//   Messages.java        Author: Lewis/Loftus
//
//   Demonstrates the use of an overridden method.
//*************************************************************

public class Messages
{
   //----------------------------------------------------------
   //  Creates two objects and invokes the message method in each.
   //----------------------------------------------------------
   public static void main(String[] args)
   {
      Thought parked = new Thought();
      Advice dates = new Advice();

      parked.message();

      dates.message();  // overridden
   }
}
```

```
//
//
//
//
//

p
{
```

```
      //-----------------------------------------------------------
      //   Creates two objects and invokes the message method in each.
      //-----------------------------------------------------------
      public static void main(String[] args)
      {
          Thought parked = new Thought();
          Advice dates = new Advice();

          parked.message();

          dates.message();   // overridden
      }
}
```

```java
//***********************************************************
//   Thought.java        Author: Lewis/Loftus
//
//   Represents a stray thought. Used as the parent of a derived
//   class to demonstrate the use of an overridden method.
//***********************************************************

public class Thought
{
   //--------------------------------------------------------
   //  Prints a message.
   //--------------------------------------------------------
   public void message()
   {
      System.out.println("I feel like I'm diagonally parked in a " +
                         "parallel universe.");

      System.out.println();
   }
}
```

```java
//****************************************************************
//  Advice.java        Author: Lewis/Loftus
//
//  Represents some thoughtful advice. Used to demonstrate the use
//  of an overridden method.
//****************************************************************

public class Advice extends Thought
{
   //-------------------------------------------------------------
   //  Prints a message. This method overrides the parent's version.
   //-------------------------------------------------------------
   public void message()
   {
      System.out.println("Warning: Dates in calendar are closer " +
                         "than they appear.");

      System.out.println();

      super.message();  // explicitly invokes the parent's version
   }
}
```

Overriding

- A method in the parent class can be invoked explicitly using the `super` reference

- If a method is declared with the `final` modifier, it cannot be overridden

- The concept of overriding can be applied to data and is called *shadowing variables*

- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overloading vs. Overriding

- Overloading deals with multiple methods with the same name in the same class, but with different signatures

- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature

- Overloading lets you define a similar operation in different ways for different parameters

- Overriding lets you define a similar operation in different ways for different object types

Quick Check

True or False?

A child class may define a method with the same name as a method in the parent.

A child class can override the constructor of the parent class.

A child class cannot override a `final` method of the parent class.

It is considered poor design when a child class overrides a method from the parent.

A child class may define a variable with the same name as a variable in the parent.

Quick Check

True or False?

A child class may define a method with the same name as a method in the parent.

True

A child class can override the constructor of the parent class.

False

A child class cannot override a `final` method of the parent class.

True

It is considered poor design when a child class overrides a method from the parent.

False

A child class may define a variable with the same name as a variable in the parent.

True, but shouldn't

Outline

Creating Subclasses

Overriding Methods

→ **Class Hierarchies**

Visibility

Designing for Inheritance

Inheritance and GUIs

The `Timer` Class

Class Hierarchies

• A child class of one parent can be the parent of another child, forming a *class hierarchy*
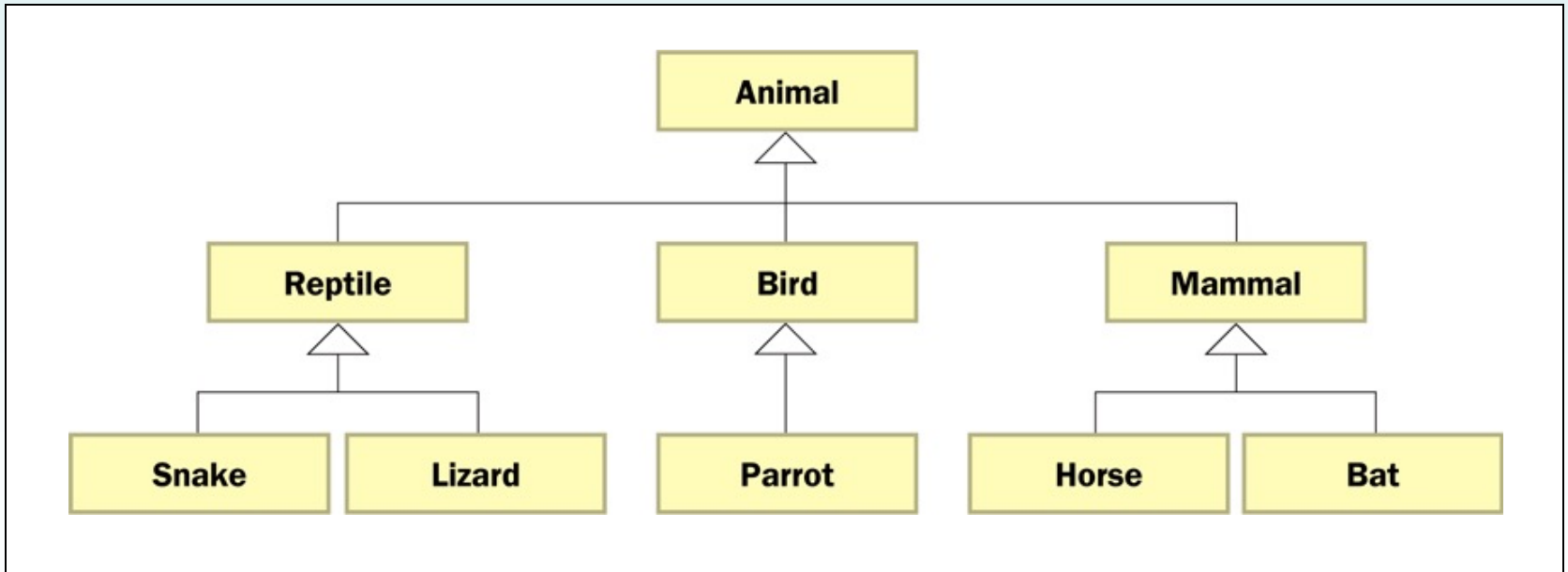
Class Hierarchies

- Two children of the same parent are called *siblings*

- Common features should be put as high in the hierarchy as is reasonable

- An inherited member is passed continually down the line

- Therefore, a child class inherits from all its ancestor classes

- There is no single class hierarchy that is appropriate for all situations

The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library

- All classes are derived from the `Object` class

- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class

- Therefore, the `Object` class is the ultimate root of all class hierarchies

The Object Class

- The `Object` class contains a few useful methods, which are inherited by all classes

- For example, the `toString` method is defined in the `Object` class

- Every time we define the `toString` method, we are actually overriding an inherited definition

- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class along with a hash code

The Object Class

- The `equals` method of the `Object` class returns true if two references are aliases

- We can override `equals` in any class to define equality in some more appropriate way

- As we've seen, the `String` class defines the `equals` method to return true if two `String` objects contain the same characters

- The designers of the `String` class have overridden the `equals` method inherited from `Object` in favor of a more useful version

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept

- An abstract class cannot be instantiated

- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Product
{
    // class contents
}
```

Abstract Classes

- An abstract class often contains abstract methods with no definitions (like an interface)

- Unlike an interface, the `abstract` modifier must be applied to each abstract method

- Also, an abstract class typically contains non-abstract methods with full definitions

- A class declared as abstract does not have to contain abstract methods -- simply declaring it as abstract makes it so

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract

- An abstract method cannot be defined as `final` or `static`

- The use of abstract classes is an important element of software design – it allows us to establish common elements in a hierarchy that are too general to instantiate

Interface Hierarchies

- Inheritance can be applied to interfaces

- That is, one interface can be derived from another interface

- The child interface inherits all abstract methods of the parent

- A class implementing the child interface must define all methods from both interfaces

- Class hierarchies and interface hierarchies are distinct (they do not overlap)

Quick Check

What are some methods defined by the `Object` class?

What is an abstract class?

Quick Check

What are some methods defined by the `Object` class?

```
String toString()
boolean equals(Object obj)
Object clone()
```

What is an abstract class?

An abstract class is a placeholder in the class hierarchy, defining a general concept and gathering elements common to all derived classes. An abstract class cannot be instantiated.

Outline

Creating Subclasses

Overriding Methods

Class Hierarchies

→ **Visibility**

Designing for Inheritance

Inheritance and GUIs

The `Timer` Class

Visibility Revisited

- It's important to understand one subtle issue related to inheritance and visibility

- All variables and methods of a parent class, even private members, are inherited by its children

- As we've mentioned, private members cannot be referenced by name in the child class

- However, private members inherited by child classes exist and can be referenced indirectly

Visibility Revisited

- Because the parent can refer to the private member, the child can reference it indirectly using its parent's methods

- The `super` reference can be used to refer to the parent class, even if no object of the parent exists

- See `FoodAnalyzer.java`
- See `FoodItem.java`
- See `Pizza.java`

```java
//***********************************************************
//   FoodAnalyzer.java        Author: Lewis/Loftus
//
//   Demonstrates indirect access to inherited private members.
//***********************************************************

public class FoodAnalyzer
{
   //--------------------------------------------------------
   //   Instantiates a Pizza object and prints its calories per
   //   serving.
   //--------------------------------------------------------
   public static void main(String[] args)
   {
      Pizza special = new Pizza(275);

      System.out.println("Calories per serving: " +
                        special.caloriesPerServing());
   }
}
```

```
//*****************************                    **********************
//   FoodAnalyzer.                 Output
//
//   Demonstrates                Calories per serving: 309    vate members.
//******************************************************************

public class FoodAnalyzer
{
   //----------------------------------------------------------------
   //   Instantiates a Pizza object and prints its calories per
   //   serving.
   //----------------------------------------------------------------
   public static void main(String[] args)
   {
      Pizza special = new Pizza(275);

      System.out.println("Calories per serving: " +
                         special.caloriesPerServing());
   }
}
```

```
//***********************************************************
//  FoodItem.java        Author: Lewis/Loftus
//
//  Represents an item of food. Used as the parent of a derived class
//  to demonstrate indirect referencing.
//***********************************************************

public class FoodItem
{
   final private int CALORIES_PER_GRAM = 9;
   private int fatGrams;
   protected int servings;

   //------------------------------------------------------------
   //  Sets up this food item with the specified number of fat grams
   //  and number of servings.
   //------------------------------------------------------------
   public FoodItem(int numFatGrams, int numServings)
   {
      fatGrams = numFatGrams;
      servings = numServings;
   }

continue
```

continue

```
    //---------------------------------------------------------
    //   Computes and returns the number of calories in this food item
    //   due to fat.
    //---------------------------------------------------------
    private int calories()
    {
       return fatGrams * CALORIES_PER_GRAM;
    }


    //---------------------------------------------------------
    //   Computes and returns the number of fat calories per serving.
    //---------------------------------------------------------
    public int caloriesPerServing()
    {
       return (calories() / servings);
    }
}
```

```
//***************************************************************
//   Pizza.java          Author: Lewis/Loftus
//
//   Represents a pizza, which is a food item. Used to demonstrate
//   indirect referencing through inheritance.
//***************************************************************

public class Pizza extends FoodItem
{
   //-----------------------------------------------------------
   //   Sets up a pizza with the specified amount of fat (assumes
   //   eight servings).
   //-----------------------------------------------------------
   public Pizza(int fatGrams)
   {
      super(fatGrams, 8);
   }
}
```

Outline

Designing for Inheritance

- As we've discussed, taking the time to create a good software design reaps long-term benefits

- Inheritance issues are an important part of an object-oriented design

- Properly designed inheritance relationships can contribute greatly to the elegance, maintainability, and reuse of the software

- Let's summarize some of the issues regarding inheritance that relate to a good software design

Inheritance Design Issues

- Every derivation should be an is-a relationship

- Think about the potential future of a class hierarchy, and design classes to be reusable and flexible

- Find common characteristics of classes and push them as high in the class hierarchy as appropriate

- Override methods as appropriate to tailor or change the functionality of a child

- Add new variables to children, but don't redefine (shadow) inherited variables

Inheritance Design Issues

- Allow each class to manage its own data; use the `super` reference to invoke the parent's constructor to set up its data

- Override general methods such as `toString` and `equals` with appropriate definitions

- Use abstract classes to represent general concepts that derived classes have in common

- Use visibility modifiers carefully to provide needed access without violating encapsulation

Restricting Inheritance

- If the `final` modifier is applied to a method, that method cannot be overridden in any derived classes

- If the `final` modifier is applied to an entire class, then that class cannot be used to derive any children at all

- Therefore, an abstract class cannot be declared as final

Outline

Creating Subclasses

Overriding Methods

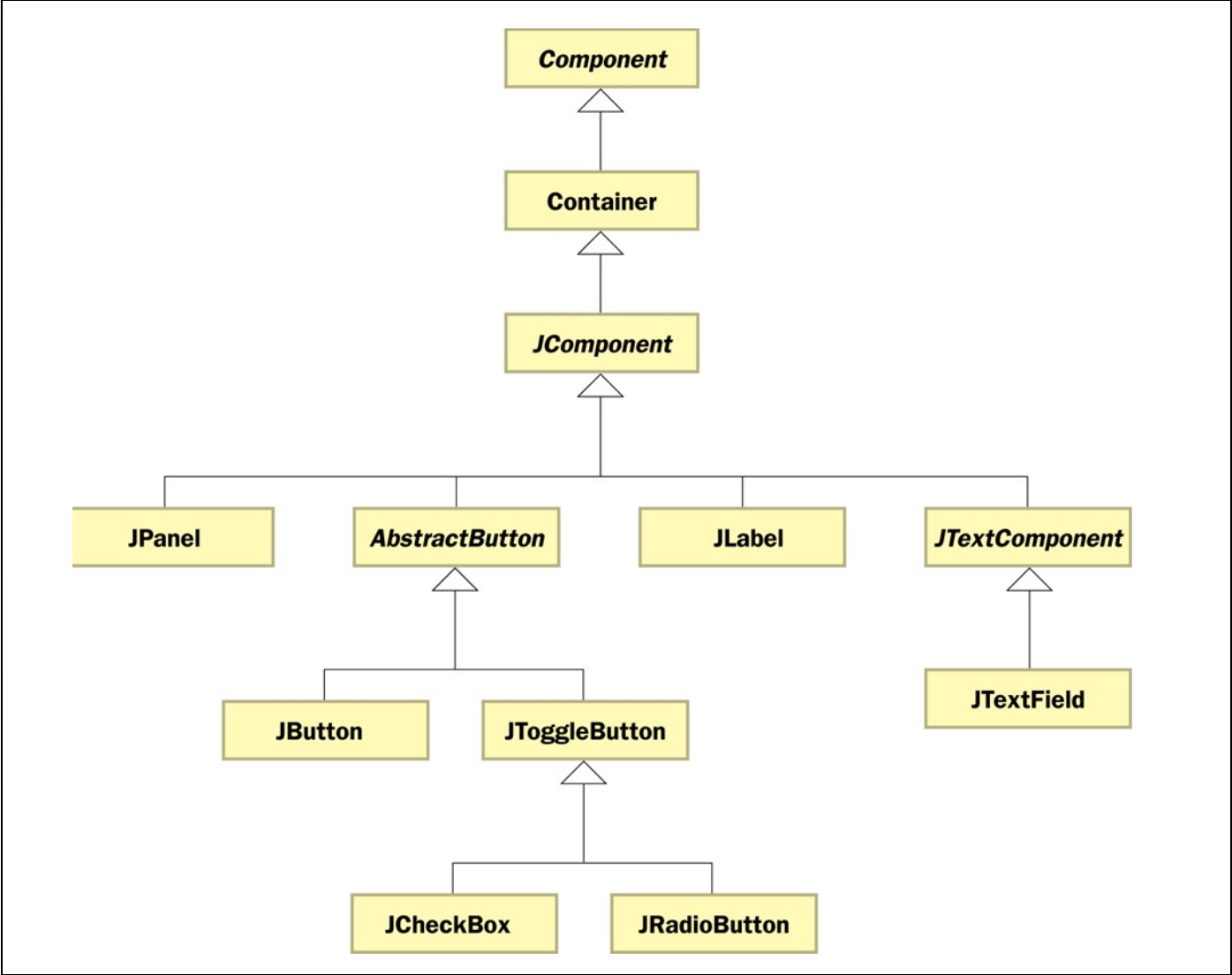**Class Hierarchies**

Visibility

Designing for Inheritance

→ **Inheritance and GUIs**

The `Timer` Class

The Component Class Hierarchy

- The Java classes that define GUI components are part of a class hierarchy

- Swing GUI components typically are derived from the `JComponent` class which is derived from the `Container` class which is derived from the `Component` class

- Many Swing components can serve as (limited) containers, because they are derived from the `Container` class

- For example, a `JLabel` object can contain an `ImageIcon`

Partial Component Class Hierarchy

The Component Class Hierarchy

- An applet is another good example of inheritance

- Recall that when we define an applet, we extend the `JApplet` class

- The `JApplet` class already handles all the details about applet creation and execution, including:

    - interaction with a Web browser

    - accepting applet parameters through HTML

    - enforcing security restrictions

The Component Class Hierarchy

- Our applet classes only have to deal with issues that specifically relate to what our particular applet will do

- When we define `paintComponent` method of an applet, we are actually overriding a method defined originally in the `JComponent` class and inherited by the `JApplet` class

Event Adapter Classes

- Inheritance also gives us a alternate technique for creating listener classes

- We've seen that listener classes can be created by implementing a particular interface, such as `MouseListener`

- We can also create a listener class by extending an *event adapter class*

- If a listener interface has more than one method, it has a corresponding adapter class, such as the `MouseAdapter` class

Event Adapter Classes

- Each adapter class implements the corresponding listener, providing empty method definitions

- When you derive a listener class from an adapter class, you only need to override the event methods that pertain to the program

- Empty definitions for unused event methods are automatically provided via inheritance

- See `OffCenter.java`
- See `OffCenterPanel.java`

```java
//*************************************************************
//  OffCenter.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an event adapter class.
//*************************************************************

import javax.swing.*;

public class OffCenter
{
    //----------------------------------------------------------
    //  Creates the main frame of the program.
    //----------------------------------------------------------
    public static void main(String[] args)
    {
        JFrame frame = new JFrame("Off Center");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.getContentPane().add(new OffCenterPanel());
        frame.pack();
        frame.setVisible(true);
    }
}
```
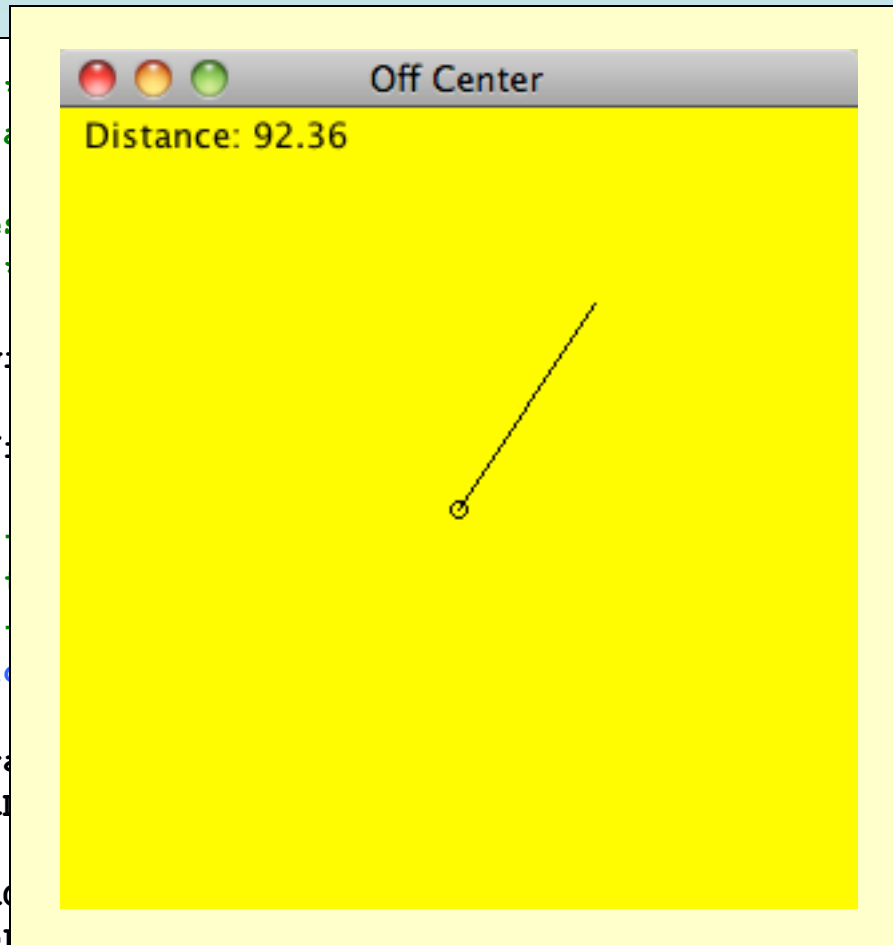
```
//****************                               ****************
//   OffCenter.j                                 
//
//   Demonstrate
//****************                               ****************

import javax.sw:

public class Of:
{
    //---------                                       --------------
    //  Creates
    //---------                                       --------------
    public stati
    {
        JFrame fra
        frame.setl                              DSE);

        frame.get(
        frame.pacl();
        frame.setVisible(true);
    }
}
```


Off Center

Distance: 92.36

```java
//****************************************************************
//  OffCenterPanel.java        Author: Lewis/Loftus
//
//  Represents the primary drawing panel for the OffCenter program.
//****************************************************************

import java.awt.*;
import java.awt.event.*;
import java.text.DecimalFormat;
import javax.swing.*;

public class OffCenterPanel extends JPanel
{
   private final int WIDTH=300, HEIGHT=300;

   private DecimalFormat fmt;
   private Point current;
   private int centerX, centerY;
   private double length;
```

continue

continue

```java
   //-----------------------------------------------------------
   //  Constructor: Sets up the panel and necessary data.
   //-----------------------------------------------------------
   public OffCenterPanel()
   {
      addMouseListener(new OffCenterListener());

      centerX = WIDTH / 2;
      centerY = HEIGHT / 2;

      fmt = new DecimalFormat("0.##");

      setPreferredSize(new Dimension(WIDTH, HEIGHT));
      setBackground(Color.yellow);
   }
```

continue

```java
    //---------------------------------------------------------
    //  Draws a line from the mouse pointer to the center point of
    //   the applet and displays the distance.
    //---------------------------------------------------------
    public void paintComponent(Graphics page)
    {
        super.paintComponent(page);

        page.setColor(Color.black);
        page.drawOval(centerX-3, centerY-3, 6, 6);

        if (current != null)
        {
            page.drawLine(current.x, current.y, centerX, centerY);
            page.drawString("Distance: " + fmt.format(length), 10, 15);
        }
    }
```

continue

```java
//*************************************************************
//   Represents the listener for mouse events. Demonstrates the
//   ability to extend an adaptor class.
//*************************************************************
private class OffCenterListener extends MouseAdapter
{
   //-----------------------------------------------------
   //  Computes the distance from the mouse pointer to the center
   //  point of the applet.
   //-----------------------------------------------------
   public void mouseClicked(MouseEvent event)
   {
      current = event.getPoint();
      length = Math.sqrt(Math.pow((current.x-centerX), 2) +
                         Math.pow((current.y-centerY), 2));
      repaint();
   }
}
}
```

Outline

The Timer Class

- The `Timer` class of the `javax.swing` package is a GUI component, but it has no visual representation

- A `Timer` object generates an action event at specified intervals

- Timers can be used to manage any events that are based on a timed interval, such as an animation

- To create the illusion of movement, we use a timer to change the scene after an appropriate delay

The Timer Class

- The `start` and `stop` methods of the `Timer` class start and stop the timer

- The delay can be set using the `Timer` constructor or using the `setDelay` method

- See `Rebound.java`
- See `ReboundPanel.java`

```java
//***********************************************************************
//   Rebound.java        Author: Lewis/Loftus
//
//   Demonstrates an animation and the use of the Timer class.
//***********************************************************************

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

public class Rebound
{
   //----------------------------------------------------------------
   //  Displays the main frame of the program.
   //----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Rebound");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new ReboundPanel());
      frame.pack();
      frame.setVisible(true);
   }
}
```

```java
//*****************                                      *************
//   Rebound.java
//
//   Demonstrates                                        lass.
//*****************                                      *************

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

public class Rebound
{
    //-------------------------------------------------------------
    //   Displays the main frame of the program.
    //-------------------------------------------------------------
    public static void main(String[] args)
    {
        JFrame frame = new JFrame("Rebound");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.getContentPane().add(new ReboundPanel());
        frame.pack();
        frame.setVisible(true);
    }
}
```
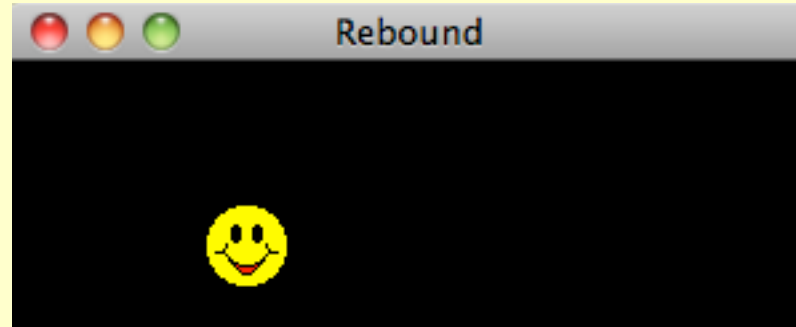
```java
//***************************************************************
//  ReboundPanel.java        Author: Lewis/Loftus
//
//  Represents the primary panel for the Rebound program.
//***************************************************************

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

public class ReboundPanel extends JPanel
{
   private final int WIDTH = 300, HEIGHT = 100;
   private final int DELAY = 20, IMAGE_SIZE = 35;

   private ImageIcon image;
   private Timer timer;
   private int x, y, moveX, moveY;

continue
```

continue

```java
    //-----------------------------------------------------------
    //  Sets up the panel, including the timer for the animation.
    //-----------------------------------------------------------
    public ReboundPanel()
    {
        timer = new Timer(DELAY, new ReboundListener());
        image = new ImageIcon("happyFace.gif");

        x = 0;
        y = 40;
        moveX = moveY = 3;

        setPreferredSize(new Dimension(WIDTH, HEIGHT));
        setBackground(Color.black);
        timer.start();
    }


    //-----------------------------------------------------------
    //  Draws the image in the current location.
    //-----------------------------------------------------------
    public void paintComponent(Graphics page)
    {
        super.paintComponent(page);
        image.paintIcon(this, page, x, y);
    }
```

continue

continue

```java
    //*********************************************************
    //  Represents the action listener for the timer.
    //*********************************************************
    private class ReboundListener implements ActionListener
    {
        //----------------------------------------------------------
        //  Updates the position of the image and possibly the direction
        //  of movement whenever the timer fires an action event.
        //----------------------------------------------------------
        public void actionPerformed(ActionEvent event)
        {
            x += moveX;
            y += moveY;

            if (x <= 0 || x >= WIDTH-IMAGE_SIZE)
               moveX = moveX * -1;

            if (y <= 0 || y >= HEIGHT-IMAGE_SIZE)
               moveY = moveY * -1;

            repaint();
        }
    }
}
```

Summary

- Chapter 9 focused on:

    - deriving new classes from existing classes
    - the `protected` modifier
    - creating class hierarchies
    - abstract classes
    - indirect visibility of inherited members
    - designing for inheritance
    - the GUI component class hierarchy
    - extending listener adapter classes
    - the `Timer` class