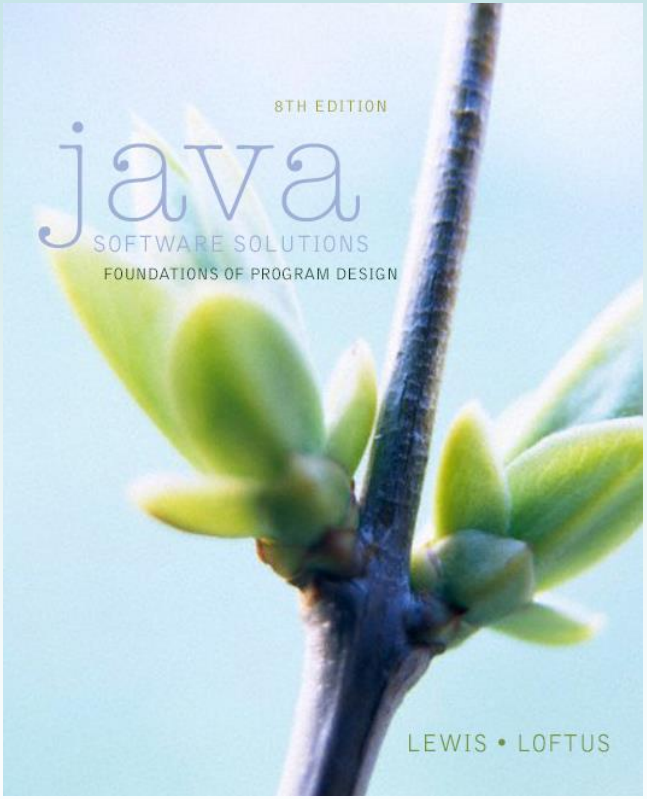
Chapter 9
Inheritance



Java Software Solutions
Foundations of Program Design
8th Edition

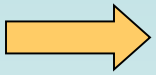
John Lewis
William Loftus

Addison-Wesley
is an imprint of
PEARSON

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**

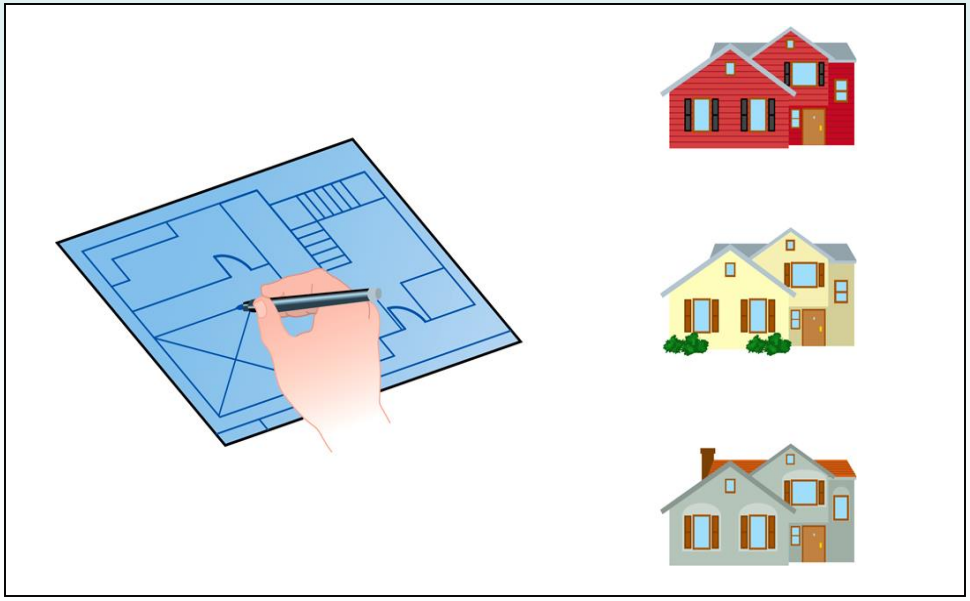
☺ **The `Timer` Class**

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

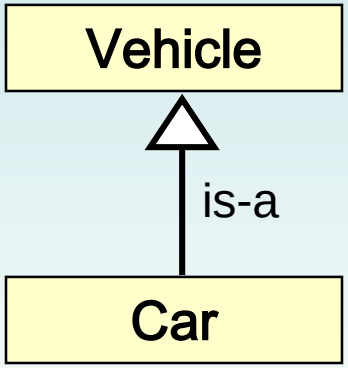
Class = Blueprint

- Basic design for a housing development might be in one blueprint (class House, the parent)
- Various designs with extra features could be based on that blueprint (class HouseWithGarage, a child) (class HouseWithDeck, another child) …



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

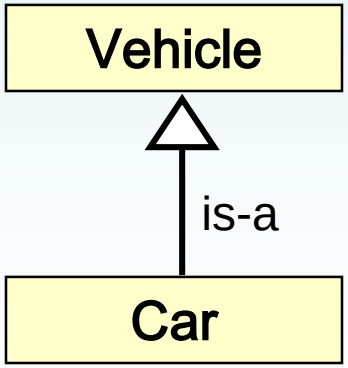
- A programmer can tailor a derived class as needed by adding new variables or methods, or by modifying the inherited ones
- Inheritance is one-way: a parent class does not change when a child inherits from it
- One benefit of inheritance is *software reuse*
- By using existing classes to create new ones, we capitalize on the design, implementation, and testing of the parent class

Deriving Subclasses

- Use `extends` to establish inheritance

```
public class Car extends Vehicle
{
    // new code for Car goes
    // here.
}
```

See `Words.java`
`Book.java`
`Dictionary.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(); // Default constructor

      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
```

```
//********************                                  ********************
//  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());
   }
}
```

```
//********************************************************************
//  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;
   }
}
```

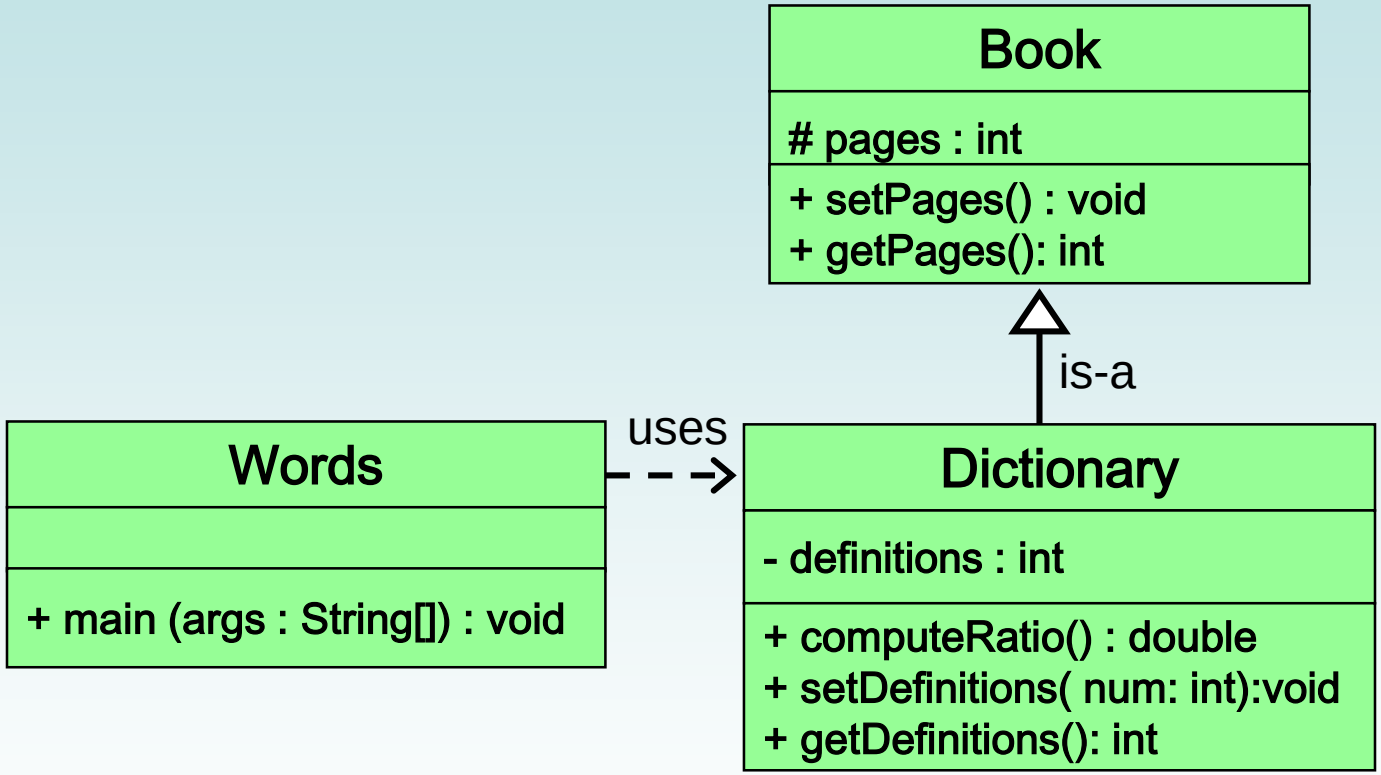
```
//********************************************************************
//  Dictionary.java       Author: Lewis/Loftus
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  inheritance.
//********************************************************************

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

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

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

Class Diagram for Words



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 used in a child class
- Variables and methods declared with public visibility can be used in a child class (and other unrelated classes)
  - but this violates encapsulation
- There is a third visibility modifier: `protected`

The protected Modifier

- Variables and methods declared with protected visibility can be used in a child class (but not all other classes)
- This provides more encapsulation than public visibility, but not as much as private visibility
- A protected variable is also visible to any class in the same package as the parent class
- Protected variables and methods have a **#** symbol preceding them in UML diagrams

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
- Use the **`super`** reference to refer to the parent class
  - often used to invoke the parent's constructor
- Do this in the first line of a child'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`

```
//********************************************************************
//  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());
   }
}
```

Output

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

```
//********************                                    ********************
//  Words2.java
//
//  Demonstrates t
//********************                                    ********************

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());
   }
}
```

```
//********************************************************************
//  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;
    }
}
```

```
//********************************************************************
//  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); // Invoke the parent's constructor

      definitions = numDefinitions;
   }
```

continue

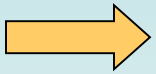
continue

```
    //-----------------------------------------------------------------
    //  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
  - Might have class `Truck` and class `Car` and want a child class that inherits from both
- 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

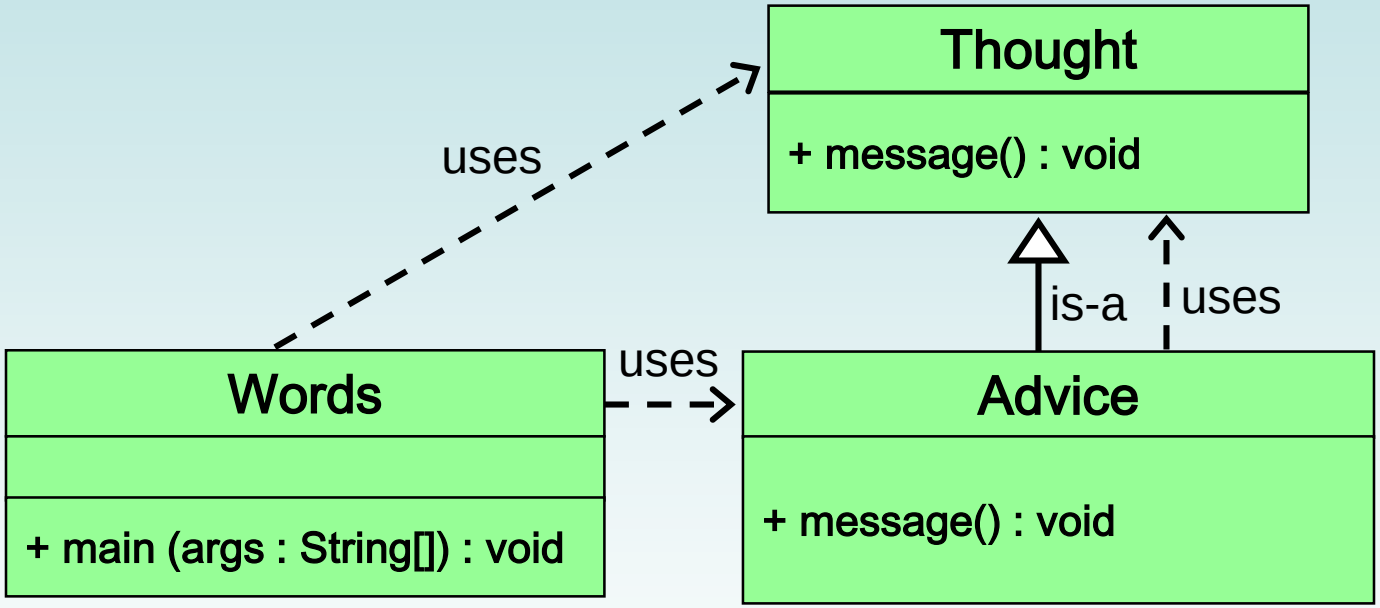
Designing for Inheritance

☺ **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`



The message() in Advice overrides the message in Thought

```
//********************************************************************
//  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(); // parent class
      Advice  dates  = new Advice();  // child class

      parked.message(); // method of parent is run

      dates.message();  // method of child is run
   }
}
```

Output

```
I feel like I'm diagonally parked in a parallel universe.

Warning: Dates in calendar are closer than they appear.

I feel like I'm diagonally parked in a parallel universe.
```

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

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

      parked.message(); // method of parent is run

      dates.message();  // method of child is run
   }
}
```

```
//********************************************************************
//  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();
   }
}
```

```
//********************************************************************
//  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*
  - a variable in a child class with the same name as a variable in the parent class hides the parent's variable in a shadow
- 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.

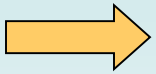
A child class may define a variable with the same name as a variable in the parent.

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

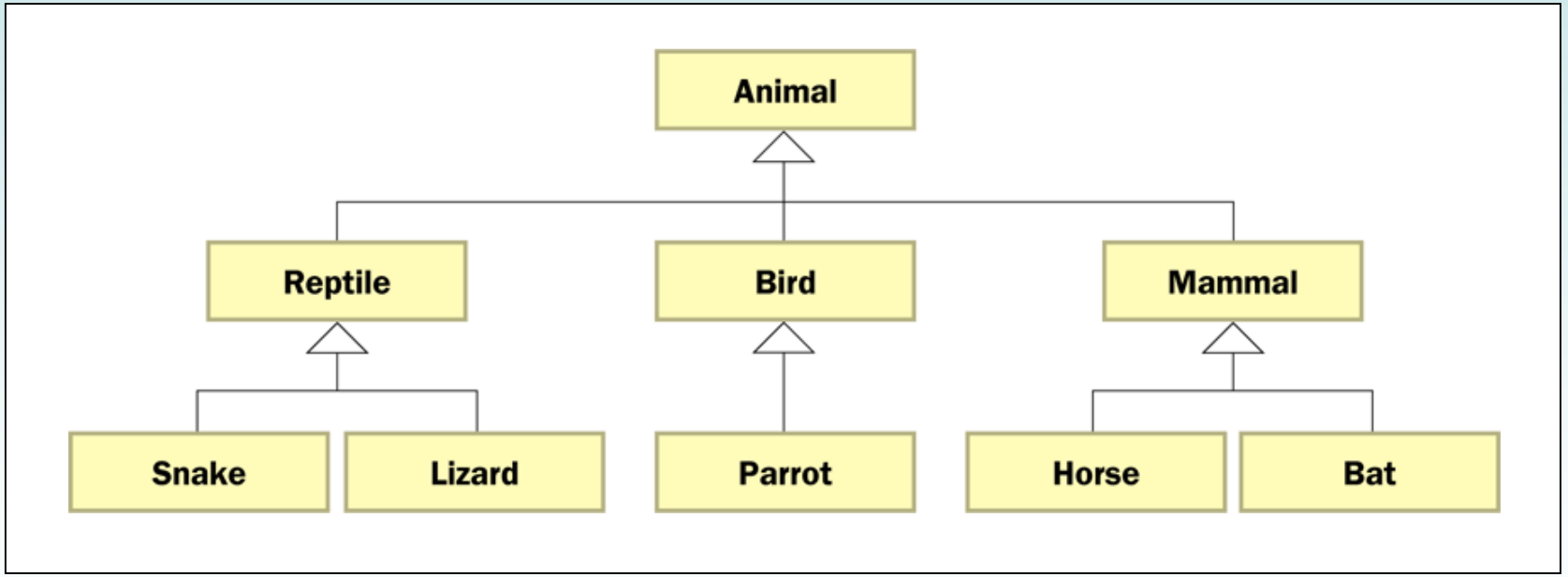
Designing for Inheritance

☺ **Inheritance and GUIs**

☺ **The `Timer` Class**

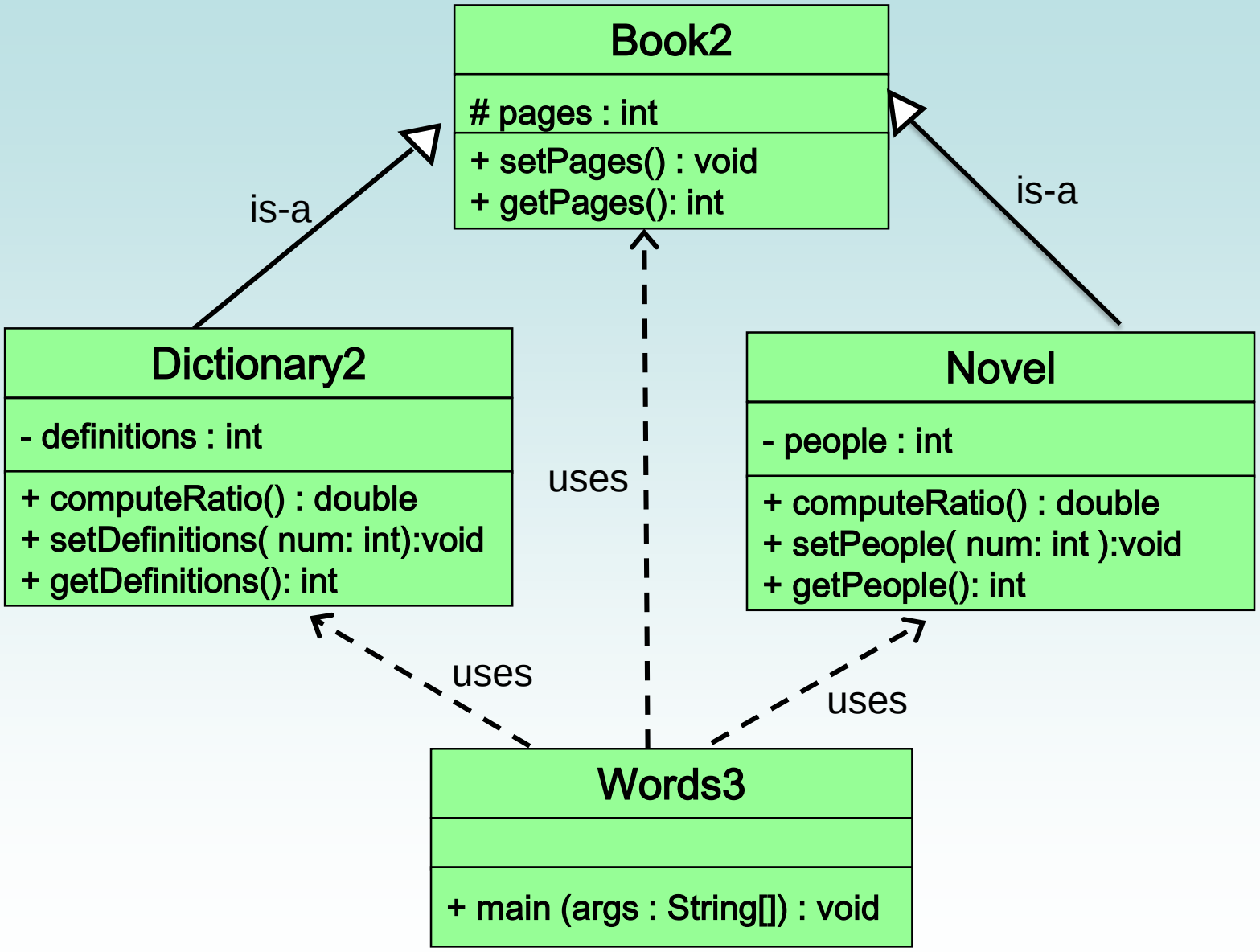
Class Hierarchies

- 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
- Children don’t inherit from each other


Book2
pages : int
+ setPages() : void
+ getPages(): int
is-a
is-a
Dictionary2
- definitions : int
+ computeRatio() : double
+ setDefinitions(num: int):void
+ getDefinitions(): int
Novel
- people : int
+ computeRatio() : double
+ setPeople(num: int):void
+ getPeople(): int
uses
uses
uses
Words3
+ main (args : String[]) : void

```
//********************************************************************
//  Words3.java       Author: Lewis/Loftus, Hacked
//********************************************************************

public class Words3
{
   public static void main(String[] args)
   {
      Book2 book = new Book2( 523 );
      System.out.println ("Number of pages:" + book.getPages() );

      Dictionary2 webster = new Dictionary2(1500, 52500);
      System.out.println("webster pages:" + webster.getPages());
      System.out.println("webster definitions:" + webster.getDefinitions());
      System.out.println("Definitions per page:" + webster.computeRatio());

      Novel dickens = new Novel( 32 );
      System.out.println("dickens pages: " + dickens.getPages());
      System.out.println("dickens people:" + dickens.getPeople());
      System.out.println("Pages per person:" + dickens.computeRatio());
   }
}
```

```
//********************************************************************
//  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;
    }
}
```

```
//********************************************************************
//  Dictionary2.java       SAME AS ABOVE
//
//  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); // Invoke the parent's constructor

      definitions = numDefinitions;
   }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  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;
    }
}
```

```
//********************************************************************
//  Novel.java
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  the use of the super reference.
//********************************************************************

public class Novel extends Book2
{
   private int people;

   public Novel(int numPages, int numPeople)
   {
      super(numPages); // Invoke the parent's constructor

      people = numPeople;
   }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Compute number of pages per person
    //-----------------------------------------------------------------
    public double computeRatio()
    {
       return (double) pages/people;
    }

    //-----------------------------------------------------------------
    //  Definitions mutator.
    //-----------------------------------------------------------------
    public void setPeople(int numPeople)
    {
       people = numPeople;
    }

    //-----------------------------------------------------------------
    //  Definitions accessor.
    //-----------------------------------------------------------------
    public int getPeople()
    {
       return poeople;
    }
}
```

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 becomes 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 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 returns 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 a way more appropriate for that class
- As we've seen, the `String` class defines the `equals` method to return true if two `String` objects contain the same characters
- The `String` class `equals` method overrides the method inherited from `Object` in favor of this more useful version

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
  - might have an abstract class Animal that includes features all animals must have
  - don't want an object that is only Animal and nothing else
  - classes derived from Animal can be instantiated
    - Dog, Cat, Spider
- An abstract class cannot be instantiated

Abstract Classes

- Use `abstract` on the class header to declare a class as abstract:

```
public abstract class Animal
{
   // 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`
- Abstract classes are important in software design
  - used to establish common features in a hierarchy that are too general to instantiate

Interface Hierarchies

- Inheritance can be applied to interfaces
  - 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

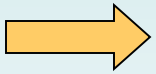
What are some methods defined by the `Object` class?

```
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

- A subtle issue related to inheritance and visibility:
- All variables and methods of a parent class, even private members, are inherited by its children
  - they are part of the child object, even though members marked private in the parent are not directly visible in the child
- Private members cannot be referenced by name in the child class
- However, private members inherited by a child exist and can be referenced indirectly

Visibility Revisited

- Because a parent can refer to its private members in a public method, the child can use the public method to indirectly access the private members
- 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`

```
//********************************************************************
//  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.
//
//  Demonstrates                                       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());
   }
}
```

Output

```
Calories per serving: 309
```

```
//********************************************************************
//  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.
//
//  A Pizza object cannot directly access its (inherited) members
//  fatGrams and servings.  But it can use its (inherited) method
//  caloriesPerServing() which can use those members.
//********************************************************************

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);
   }
}
```

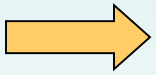
Outline

Creating Subclasses

Overriding Methods

Class Hierarchies

Visibility



Designing for Inheritance

☺ **Inheritance and GUIs**

☺ **The `Timer` Class**

Designing for Inheritance

- Creating 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

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

- The **`final`** modifier says that a method cannot be overridden in any derived classes
- The **`final`** modifier applied to an entire class says that the class cannot be used to derive any children at all
- Therefore, an abstract class cannot be declared as final

Outline

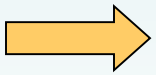
Creating Subclasses

Overriding Methods

Class Hierarchies

Visibility

Designing for Inheritance



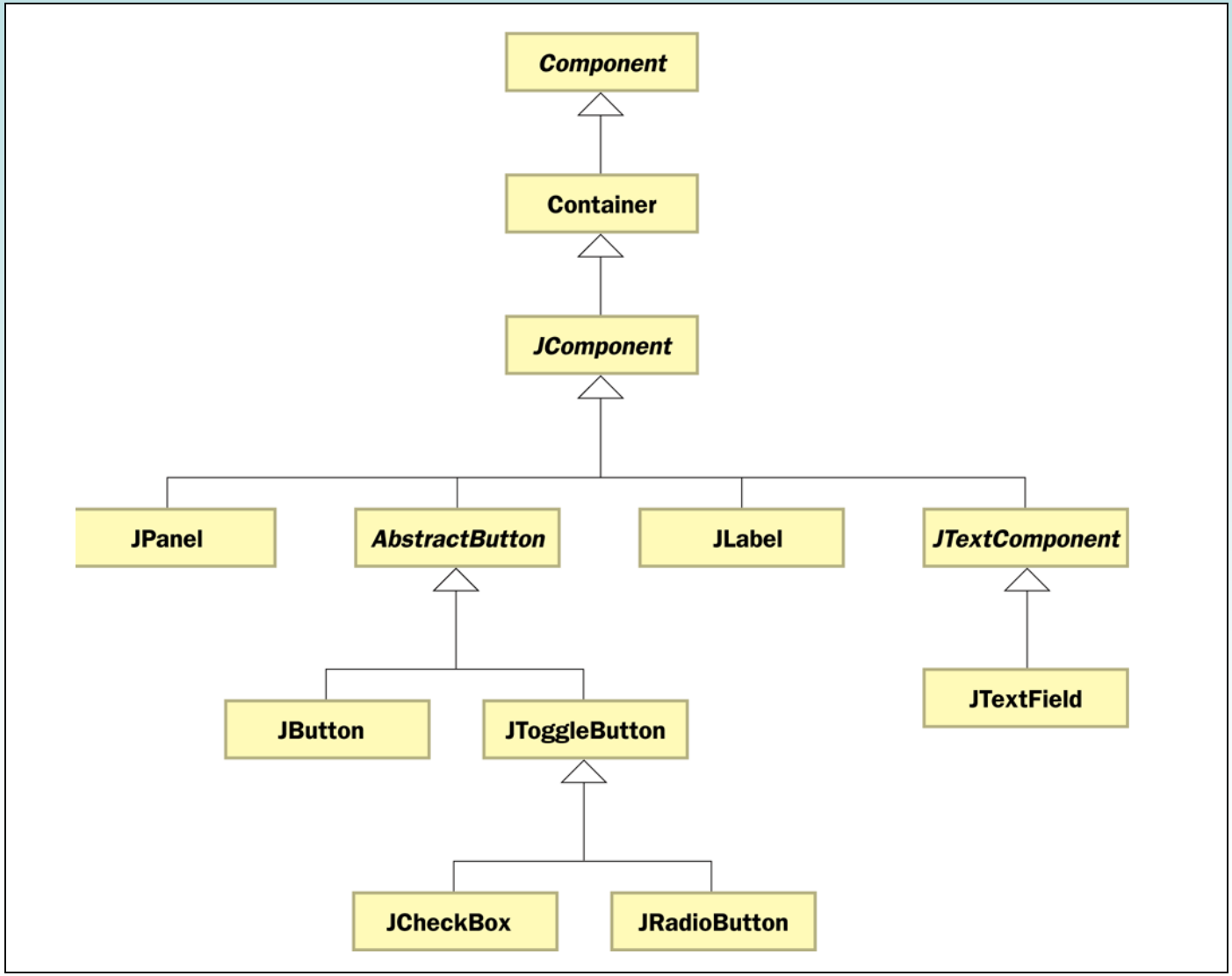
☺ Inheritance and GUIs

☺ 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

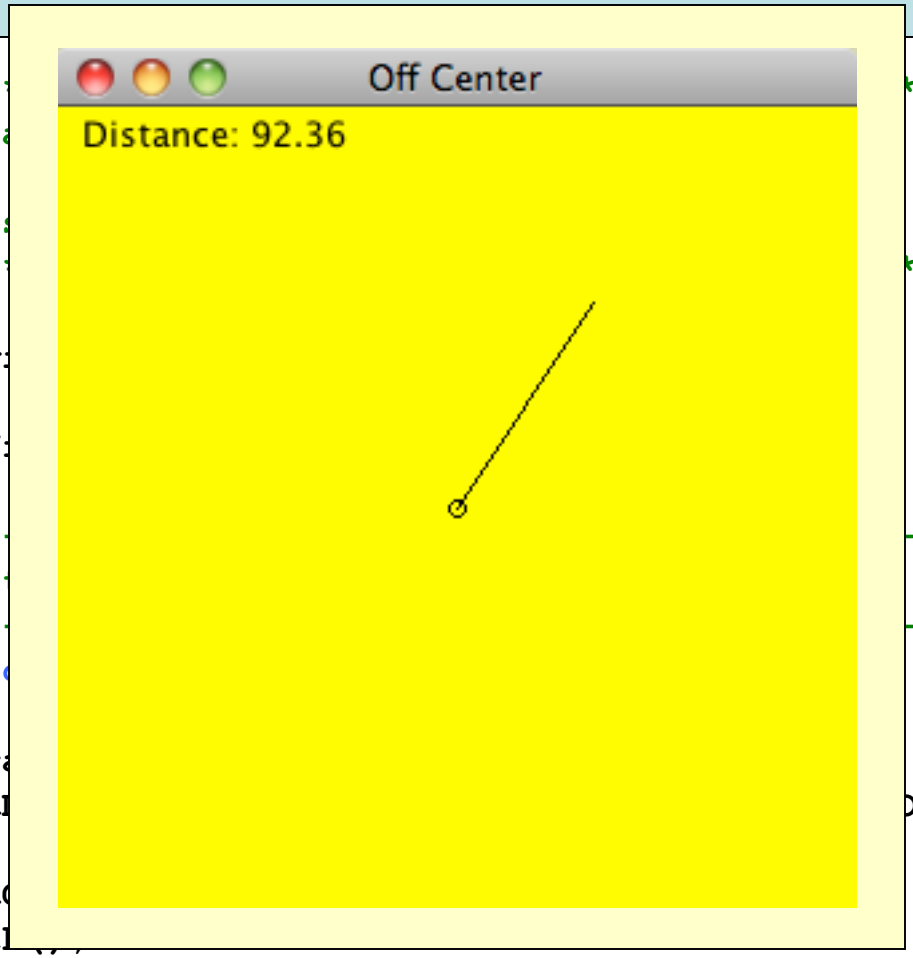
- Inheritance also gives us a alternate technique for creating listener classes
- We've seen that listener classes can be created by implementing a particular interface, such as `MouseListener`
- We can also create a listener class by extending an *event adapter class*
- If a listener interface has more 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`

```
//********************************************************************
//  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.ja
//
//  Demonstrates
//*************

import javax.swi

public class Off
{
   //----------
   //  Creates
   //----------
   public stati
   {
      JFrame fra
      frame.setL

      frame.getC
      frame.pack
      frame.setVisible(true);
   }
}
```



```
//********************************************************************
//  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

```
   //-----------------------------------------------------------------
   //  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

continue

```
//-----------------------------------------------------------------
//  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

continue

```
    //*****************************************************************
    //  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

Creating Subclasses

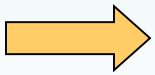
Overriding Methods

Class Hierarchies

Visibility

Designing for Inheritance

☺ Inheritance and GUIs



☺ The `Timer` Class

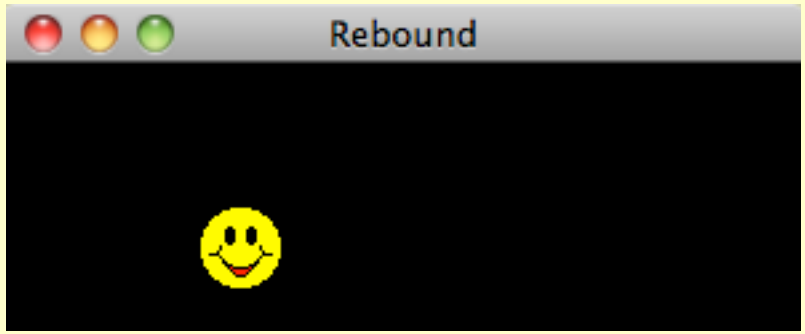
☺ The Timer Class

- 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 manage events that are based on a time interval, such as an animation
- To create the illusion of movement, 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`

```
//********************************************************************
//  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);
   }
}
```




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

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);
   }
}
```

```
//********************************************************************
//  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

```
    //-----------------------------------------------------------------
    //  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

```
    //*****************************************************************
    //  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