# Inheritance and Class Hierarchies

Based on Koffmann and Wolfgang

Chapter 3

# Chapter Outline

- ***Inheritance*** and how it facilitates code reuse
- How does Java ***find the "right" method*** to execute?
  - (When more than one has the same name ...)
- Defining and using ***abstract classes***
- Class **`Object`**: its methods and how to override them
- How to "clone" an object
- The difference between:
  - A ***true clone*** (deep copy) and
  - A ***shallow copy***

# Chapter Outline (2)

- Why Java does ***not*** implement *multiple inheritance*
- Get some of the advantages of multiple inheritance:
  - *Interfaces*
  - *Delegation*
- Sample class hierarchy: *drawable shapes*
- An *object factory* and how to use it
- Creating *packages*
  - Code *visibility*

# Inheritance and Class Hierarchies

- Object-oriented programming (OOP) is popular because:
  - It enables *reuse* of previous code saved as *classes*

- All Java classes are arranged in a hierarchy
  - **`Object`** is the *superclass* of all Java classes

- *Inheritance* and hierarchical organization capture idea:
  - One thing is a *refinement* or *extension* of another

# Inheritance and Class Hierarchies (2)

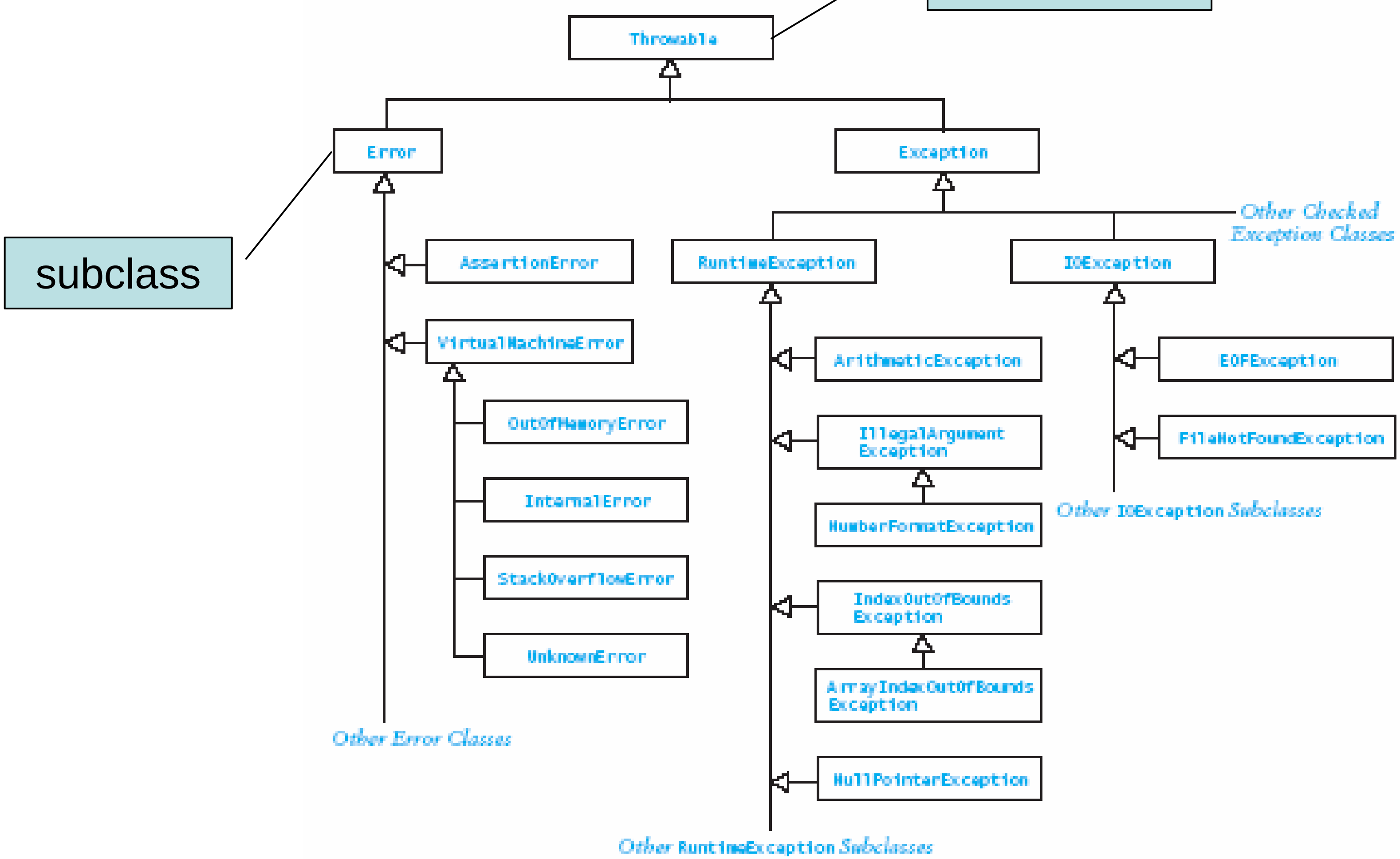



**FIGURE 3.1**
Summary of Exception Class Hierarchy

# *Is-a* Versus *Has-a* Relationships

- Confusing *has-a* and *is-a* leads to misusing inheritance
- Model a *has-a* relationship with an *attribute* (variable)

```
public class C { ... private B part; ...}
```

- Model an *is-a* relationship with inheritance
  - If every **C** is-a **B** then model **C** as a subclass of **B**
  - Show this: in **C** include **extends B**:

```
public class C extends B { ... }
```

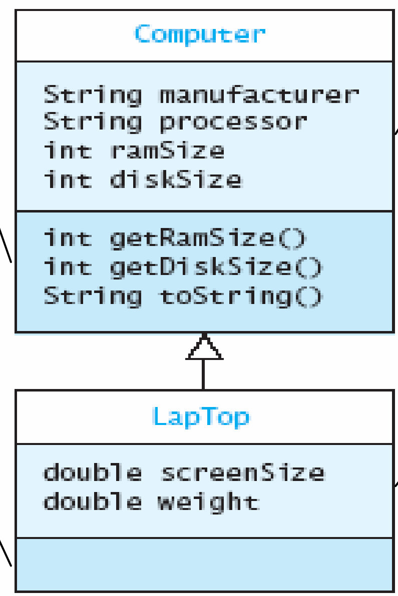
# A Superclass and a Subclass

- Consider two classes: **`Computer`** and **`Laptop`**
- A laptop is a *<u>kind</u>* of computer: therefore a subclass

FIGURE 3.2
Classes LapTop and Computer

methods of **`Computer`** *and all subclasses*

variables of **`Computer`** *and all subclasses*




*additional* Methods for class **`Laptop`** (and its subclasses)

*additional* variables for class **`Laptop`** (and its subclasses)

# Illustrating *Has-a* with **`Computer`**

```
public class Computer {
  private Memory mem;
  ...
}

public class Memory {
  private int size;
  private int speed;
  private String kind;
  ...
}
```

A **`Computer`** has only one **`Memory`**

But neither *is-a* the other

# Initializing Data Fields in a Subclass

- What about data fields of a superclass?
  - Initialize them by invoking a superclass constructor with the appropriate parameters
- If the subclass constructor skips calling the superclass ...
  - Java automatically calls the *no-parameter* one
- ***Point:*** Insure superclass fields initialized *before* subclass starts to initialize its part of the object

# Example of Initializing Subclass Data

```
public class Computer {
  private String manufacturer;  ...
  public Computer (String manufacturer, ...) {
    this.manufacturer = manufacturer; ...
  }
}

public class Laptop extends Computer {
  private double weight; ...
  public Laptop (String manufacturer, ...,
                 double weight, ...) {
    super(manufacturer, ...);
    this.weight = weight;
  }
}
```

# **`Protected`** Visibility for Superclass Data

- **`private`** data are *not accessible* to subclasses!
- **`protected`** data fields *accessible in subclasses*

  (Technically, accessible in *same package)*

- Subclasses often written by others, and
- Subclasses should avoid relying on superclass details

- ***So …*** in general, **`private`** is better

# Method *Overriding*

- If subclass has a method of a superclass (same signature), that method *overrides* the superclass method:

```
public class A { ...
   public int M (float f, String s) { bodyA }
}

public class B extends A { ...
   public int M (float f, String s) { bodyB }
}
```

- If we call **M** on an instance of **B** (or subclass of **B**), **bodyB** runs
- In **B** we can access **bodyA** with: **super.M(...)**
- The subclass **M** must have same return type as superclass **M**

# Method *Overloading*

- **Method overloading:** *multiple* methods ...
  - With the *same name*
  - But *different signatures*
  - In the *same class*
- Constructors are often overloaded
- Example:
  - **`MyClass (int inputA, int inputB)`**
  - **`MyClass (float inputA, float inputB)`**

# Example of Overloaded Constructors

```
public class Laptop extends Computer {
  private double weight; ...
  public Laptop (String manufacturer,
                 String processor, ...,
                 double weight, ...) {
    super(manufacturer, processor, ...);
    this.weight = weight;
  }
  public Laptop (String manufacturer, ...,
                 double weight, ...) {
    this(manufacturer, “Pentium”, ...,
         weight, ...);
  }
}
```

# Overloading Example From Java Library

**`ArrayList`** has two **`remove`** methods:

**`remove (int position)`**

- Removes object that is at a specified *place* in the list

**`remove (Object obj)`**

- Removes a *specified object* from the list

It also has two **`add`** methods:

**`add (Element e)`**

- Adds new object to the *end* of the list

**`add (int index, Element e)`**

- Adds new object at a *specified place* in the list

# Polymorphism

- Variable of *superclass type* can refer to *object of subclass type*
- *Polymorphism* means “many forms” or “many shapes”
- Polymorphism lets the JVM determine *at run time* which method to invoke
- *At compile time:*
  - Java compiler cannot determine exact type of the object
  - But it *is* known at run time
- Compiler knows enough for safety: the *attributes* of the type
  - *Subclasses guaranteed to obey*

# Interfaces vs Abstract Classes vs Concrete Classes

- A Java *interface* can declare methods
  - But cannot implement them
  - Methods of an interface are called *abstract methods*
- An *abstract **class*** can have:
  - Abstract methods (no body)
  - Concrete methods (with body)
  - Data fields
- Unlike a concrete class, an *abstract class* ...
  - *Cannot be instantiated*
  - *Can declare abstract methods*
    - Which *must* be implemented in all *concrete* subclasses

# Abstract Classes and Interfaces

- Abstract classes and interfaces cannot be instantiated

- An abstract class *can* have constructors!
  - *Purpose:* initialize data fields when a subclass object is created
  - Subclass uses **`super(…)`** to call the constructor

- An abstract class may *implement* an interface
  - But need not define all methods of the interface
  - Implementation of them is left to subclasses

# Example of an Abstract Class

```
public abstract class Food {
  public final String name;
  private double calories;
  public double getCalories () {
    return calories;
  }
  protected Food (String name, double calories) {
    this.name     = name;
    this.calories = calories;
  }
  public abstract double percentProtein();
  public abstract double percentFat();
  public abstract double percentCarbs();
}
```

# Example of a Concrete Subclass

```
public class Meat extends Food {
  private final double protCal; ...;
  public Meat (String name, double protCal,
               double fatCal double carbCal) {
    super(name, protCal+fatCal+carbCal);
    this.protCal = protCal;
    ...;
  }
  public double percentProtein () {
    return 100.0 * (protCal / getCalories());
  }
  ...;
}
```

# Example: **`Number`** and the Wrapper Classes

Declares what the (concrete) subclasses have in common

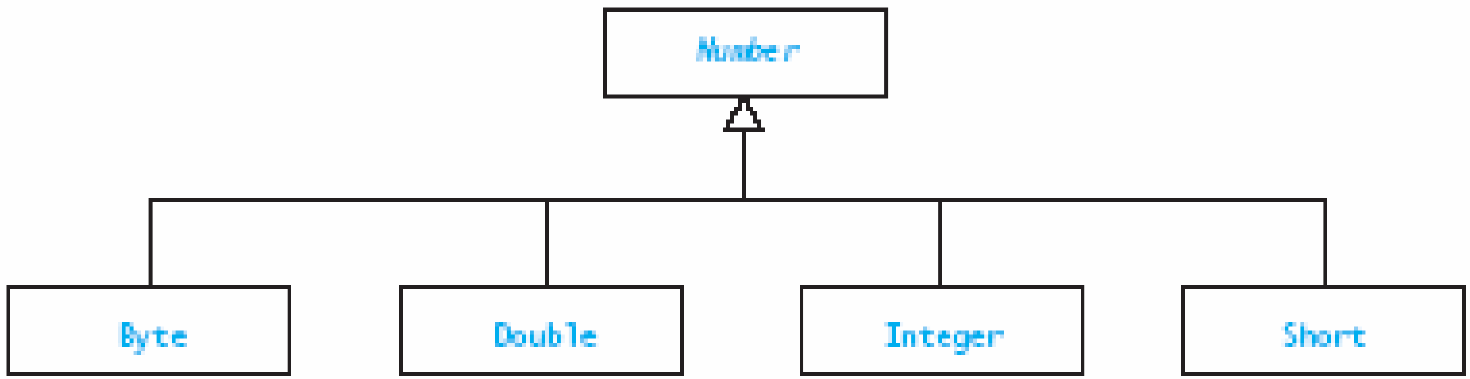



**FIGURE 3.4**
The Abstract Class `java.lang.Number` and Some of Its Subclasses

# Inheriting from Interfaces vs Classes

- A class can *extend* 0 or 1 superclass
  - Called *single inheritance*
- An interface cannot extend a class at all
  - (Because it is not a class)
- A class or interface can *implement* 0 or more interfaces
  - Called *multiple inheritance*

# Summary of Features of Actual Classes, Abstract Classes, and Interfaces

**TABLE 3.1**

Comparison of Actual Classes, Abstract Classes, and Interfaces

| Property | Actual Class | Abstract Class | Interface |
|---|---|---|---|
| Instances (objects) of this can be created | Yes | No | No |
| This can define instance variables and methods | Yes | Yes | No |
| This can define constants | Yes | Yes | Yes |
| The number of these a class can extend | 0 or 1 | 0 or 1 | 0 |
| The number of these a class can implement | 0 | 0 | Any number |
| This can extend another class | Yes | Yes | No |
| This can declare abstract methods | No | Yes | Yes |
| Variables of this type can be declared | Yes | Yes | Yes |

# Class **`Object`**

- **`Object`** is the root of the class hierarchy
  - Every *class* has **`Object`** as a superclass
- All classes inherit the methods of Object
  - But may override them

**TABLE 3.2**
Methods of Class `java.lang.Object`

| Method | Behavior |
|---|---|
| `Object clone()` | Makes a copy of an object. |
| `boolean equals(Object obj)` | Compares this object to its argument. |
| `int hashCode()` | Returns an integer hash code value for this object. |
| `String toString()` | Returns a string that textually represents the object. |

# The Method **`toString`**

- You should always override **`toString`** method if you want to print object state

- If you do *not* override it:
  - **`Object.toString`** will return a **`String`**
  - Just not the **`String`** you want!

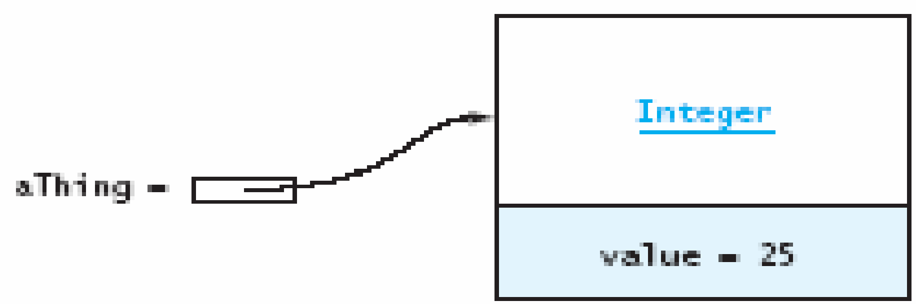
    Example: **`ArrayBasedPD@ef08879`**

    ... The name of the class, `@`, instance's hash code

# Operations Determined by Type of Reference Variable

- Variable can refer to object whose type is a ***subclass*** of the variable's declared type
- Type of the ***variable*** determines what operations are legal
- Java is ***strongly typed*** **`Object athing = new Integer(25);`**
  - Compiler always verifies that variable's type includes the class of every expression assigned to the variable

**FIGURE 3.5**
Type Integer Object Referenced by aThing (type Object)



# Casting in a Class Hierarchy

- *Casting* obtains a reference of different, but *matching,* type
- Casting *does not change* the object!
  - It creates an anonymous reference to the object

```
Integer aNum = (Integer)aThing;
```

- *Downcast:*
  - Cast *superclass* type to *subclass* type
  - Checks *at run time* to make sure it's ok
  - If not ok, throws **`ClassCastException`**

# Casting in a Class Hierarchy (2)

- **`instanceof`** can guard against **`ClassCastException`**

```
Object obj = ...;
if (obj instanceof Integer) {
  Integer i = (Integer)obj;
  int val = i.intValue();
  ...;
} else {
  ...
}
```

# Downcasting From an Interface Type

```
Collection c = new ArrayList();
...;
... ((ArrayList)c).get(3) ...
```

# Polymorphism Reduces Need For Type Tests

```
// Non OO style:
if (stuff[i] instanceof Integer)
  sum += ((Integer) stuff[i]).doubleValue();
else if (stuff[i] instanceof Double)
  sum += ((Double) stuff[i]).doubleValue();
...

// OO style:
sum += stuff[i].doubleValue();
```

# Polymorphism and Type Tests (2)

- Polymorphic code style is more *extensible*
  - Works *automatically* with new subclasses
- Polymorphic code is more *efficient*
  - System does one indirect branch vs many tests

- ***So ...*** uses of **`instanceof`** are <u>*suspect*</u>

# Java 5.0 Reduces Explicit Conversions

- Java 1.4 and earlier:

```
Character ch = new Character(‘x’);
char nextCh = ch.charValue();
```

- Java 5.0:

```
Character ch = ‘x’;       // called auto-box
char nextCh = ch;         // called auto-unbox
```

- Java 5.0 generics also reduce explicit casts

# The Method **`Object.equals`**

- **`Object.equals`** method has parameter of type **`Object`**

  **`public boolean equals (Object other) { ... }`**

- Compares two objects to determine if they are equal
- Must override **`equals`** in order to support comparison

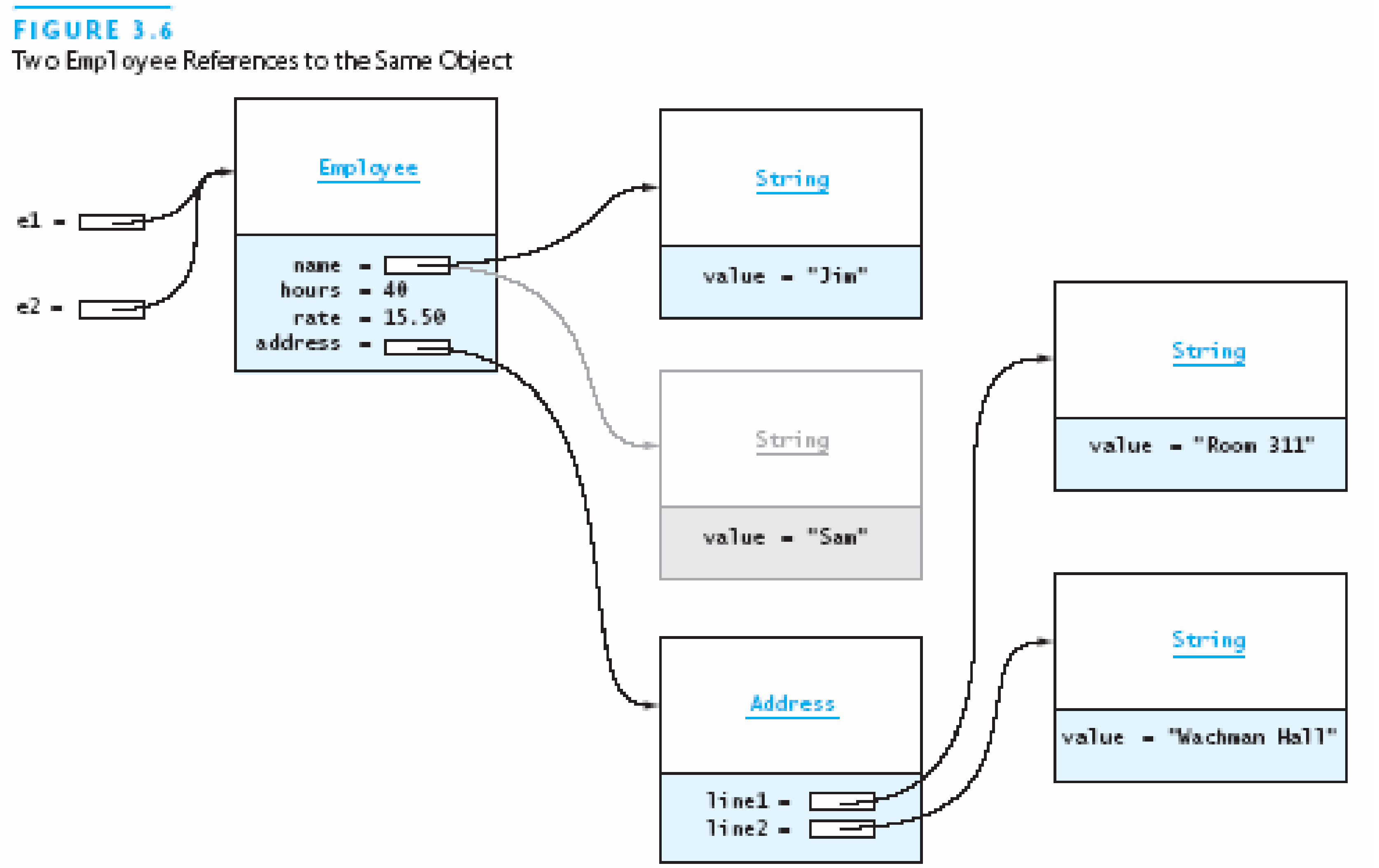
# Cloning

- Purpose analogous to cloning in biology:
  - Create an *independent copy* of an object

- Initially, objects and clone store *same information*

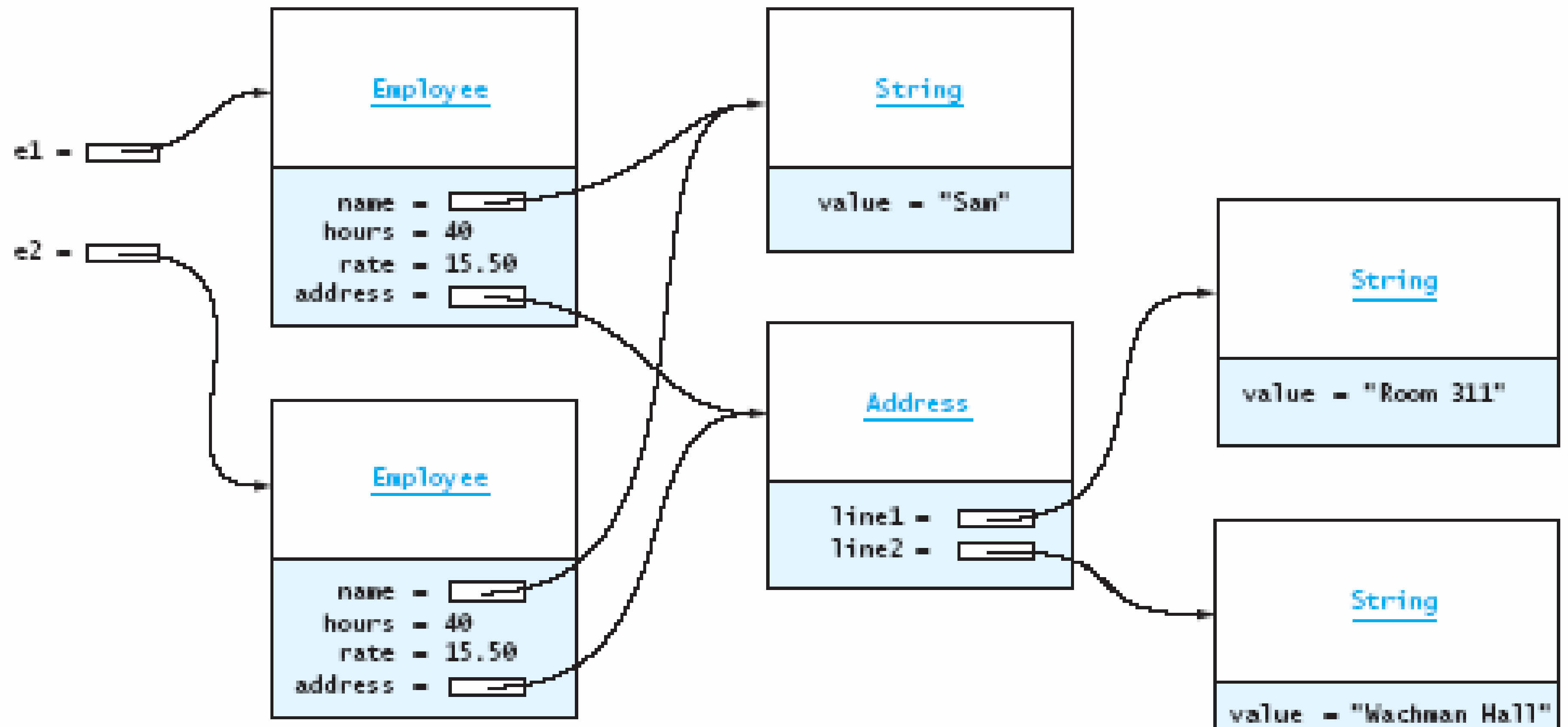
- You can change one object *without affecting the other*

# The Shallow Copy Problem (Before)



# The Shallow Copy Problem (After)

FIGURE 3.7
An Employee Object and a Shallow Copy

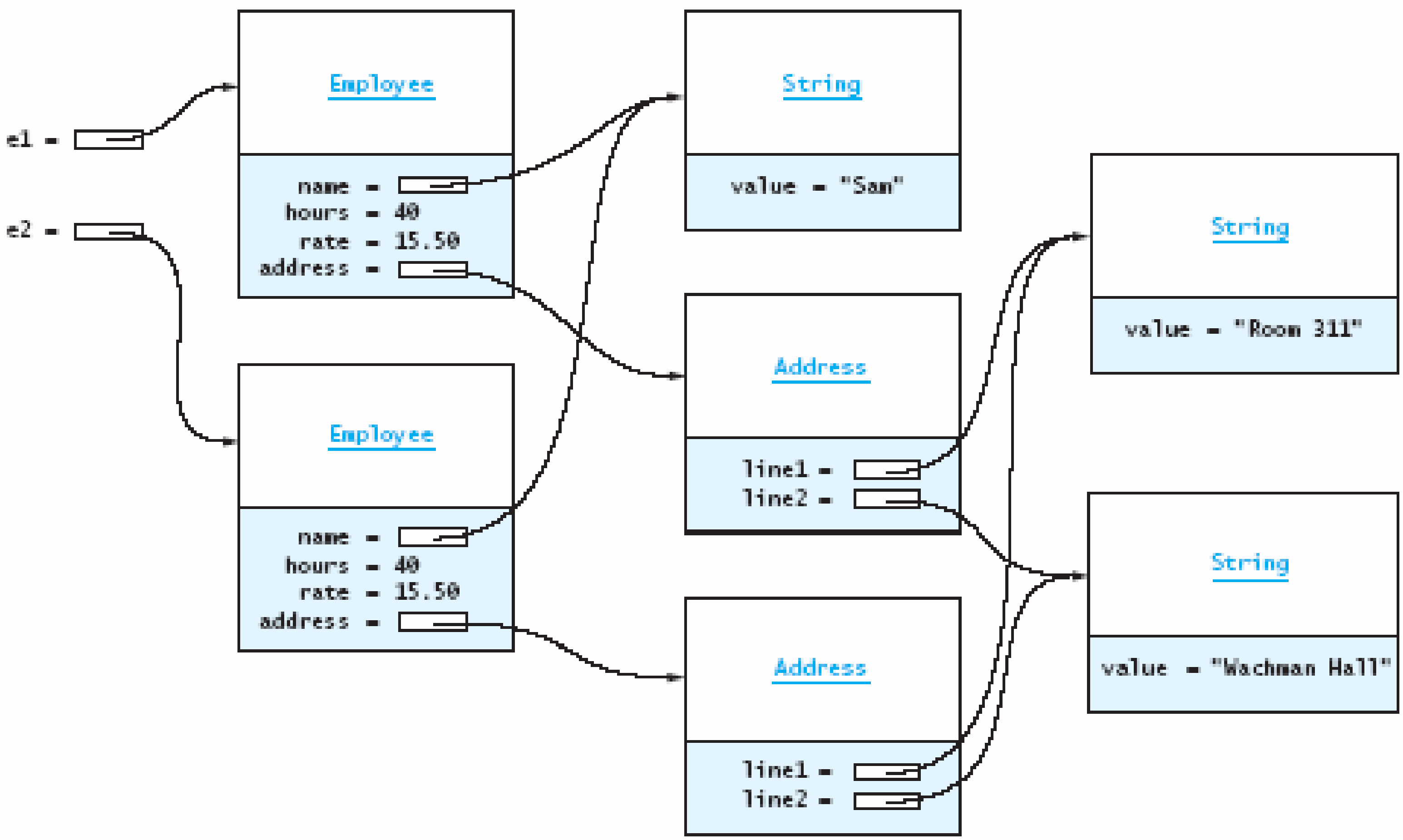


# The **`Object.clone`** Method

- **`Object.clone`** addresses the shallow copy problem
- The initial copy is a shallow copy, but ...
- For a *deep copy:*
  - Create cloned copies of all components by ...
  - Invoking *their* respective clone methods

# The **`Object.clone`** Method (2)

**FIGURE 3.8**
Deep Copy or Clone of an Object



# The **`Object.clone`** Method (3)

```
public class Employee implements Cloneable {
  ...
  public Object clone () {
    try {
      Employee cloned = (Employee)super.clone();
      cloned.address = (Address)address.clone();
      return cloned;
    } catch (CloneNotSupportedException e) {
      throw new InternalError();
    }
  }
}
```

# The **`Object.clone`** Method (4)

```
public class Address implements Cloneable {
  ...
  public Object clone () {
    try {
      Address cloned = (Address)super.clone();
      return cloned;
    } catch (CloneNotSupportedException e) {
      throw new InternalError();
    }
  }
}
```

# The **`Object.clone`** Method (5)

```
Employee[] company = new Employee[10];
...
Employee[] newCompany =
    (Employee[])company.clone();
// need loop below for deep copy
for (int i = 0; i < newCompany.length; i++) {
  newCompany[i] =
      (Employee)newCompany[i].clone();
}
```

# Multiple Inheritance, Multiple Interfaces, and Delegation

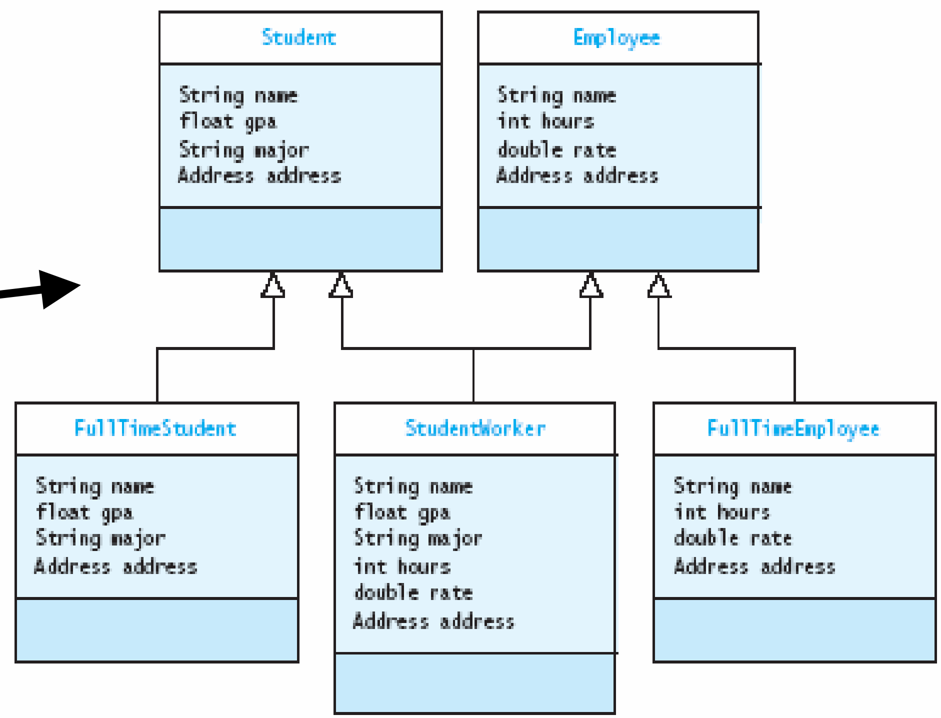
- *Multiple inheritance:* the ability to *extend* more than one class
- Multiple inheritance ...
  - Is difficult to implement efficiently
  - Can lead to ambiguity: if two parents implement the same method, which to use?
  - Therefore, Java does not allow a class to extend more than one class

# Multiple Interfaces can Emulate Multiple Inheritance

- A class can implement two or more interfaces
- Multiple interfaces emulate multiple inheritance

Desired, but illegal, situation

FIGURE 3.9
Class StudentWorker Extends Student and Employee

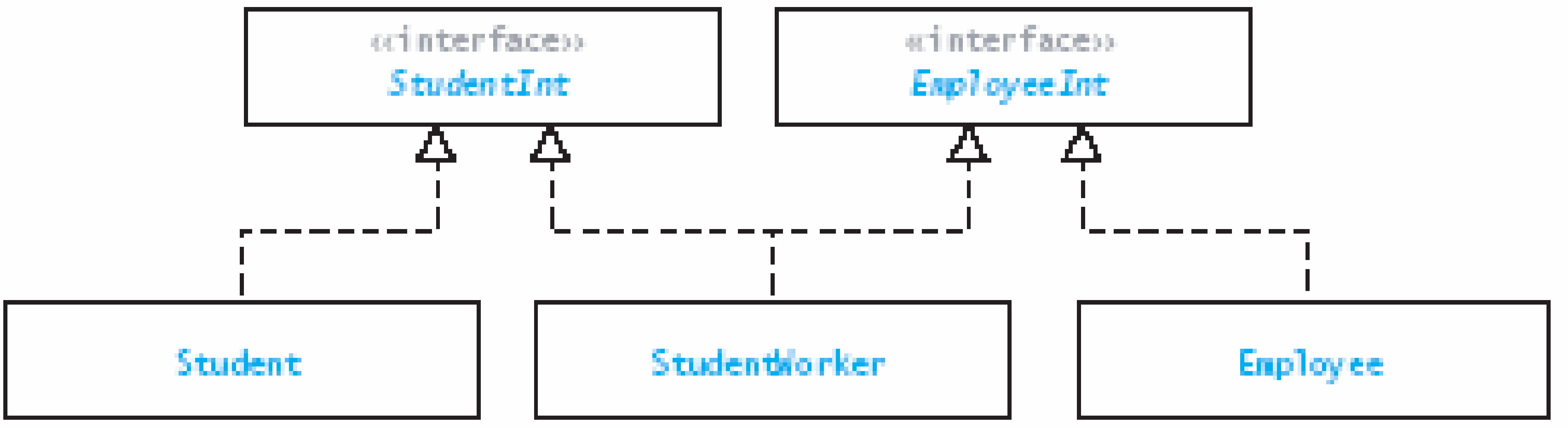


# Multiple Interfaces can Emulate Multiple Inheritance

- Approximating the desire with interfaces:

**FIGURE 3.10**

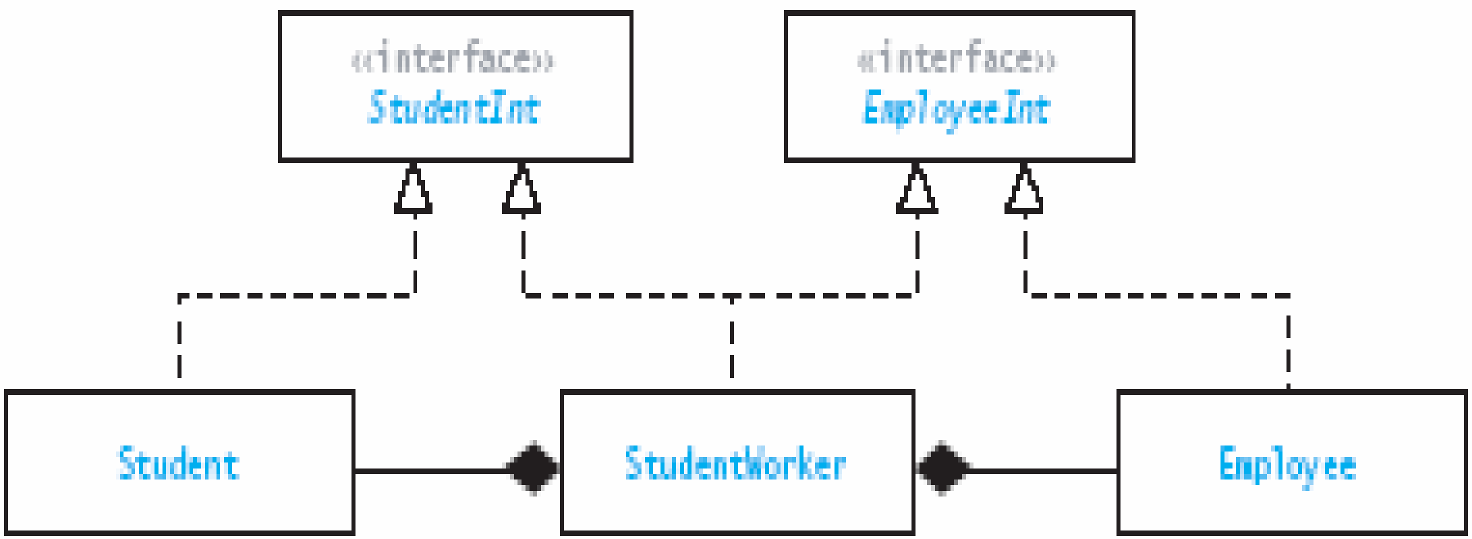
Class Hierarchy with Interfaces StudentInt and EmployeeInt



# Supporting Reuse Using *Delegation*

- Reduce "cut and paste polymorphism": copied code
- *Idea:* Object of another class does the work
- *Delegation:* original object ***delegates*** to the other

**FIGURE 3.11**
UML Diagram with Delegation



# Delegation: Implementing It

- Class **`StudentWorker`** implements interfaces **`StudentInt`** and **`EmployeeInt`**
- Class **`StudentWorker`** has-a **`Student`** and has-an **`Employee`**
- **`StudentWorker`** implements (some) **`StudentInt`** methods with calls to its **`Student`** object
- Likewise for **`EmployeeInt`** methods
- **`StudentWorker`** implements **`getName()`** itself, etc.

# Delegation: More About It

- Delegation is like applying hierarchy ideas to instances rather than classes
- There have been whole OO languages based more on delegation than on classes
- *Opinion:* Classes are better, when they can do what you need
- Downside of delegation: Not as efficient, because of level of indirection, and need for separate objects

# Packages and Directories

- A Java ***package*** is a group of *cooperating classes*
- Java programs are organized into packages
- The Java API is also organized as packages
- Indicate the package of a class at the top of the file:

```
package thePackageForThisClass;
```

- Classes of the *same package* should be in the *same directory* (folder)
- Classes in the *same folder* must be in the *same package*

# Packages and Visibility

- Classes *<u>not</u>* part of a package can access only **`public`** members of classes in the package
- The default visibility is *<u>package visbility</u>*
  - Has no keyword: indicate by not using another
  - Others are: **`public`**, **`protected`**, **`private`**
- Package visibility: between **`private`** and **`protected`**
  - Items with package visibility: visible in package, invisible outside package
  - Items with protected visibility: visible in package *<u>and</u>* in subclasses outside the package

# The No-Package-Declared Environment

- There is a default package
  - It contains files that have no package declared
- Default package ok for small projects
  - Packages good for larger groups of classes

# Visibility Supports Encapsulation

- Visibility rules enforce encapsulation in Java
- **`private`**: Good for members that should be invisible even in subclasses
- package: Good to shield classes and members from classes outside the package
- **`protected`**: Good for visibility to extenders of classes in the package
- **`public`**: Good for visibility to all

# Visibility Supports Encapsulation (2)

- Encapsulation provides insulation against change
- Greater visibility means less encapsulation

- ***So:*** use minimum visibility possible for getting the job done!

# Visibility Supports Encapsulation (3)

**TABLE 3.3**

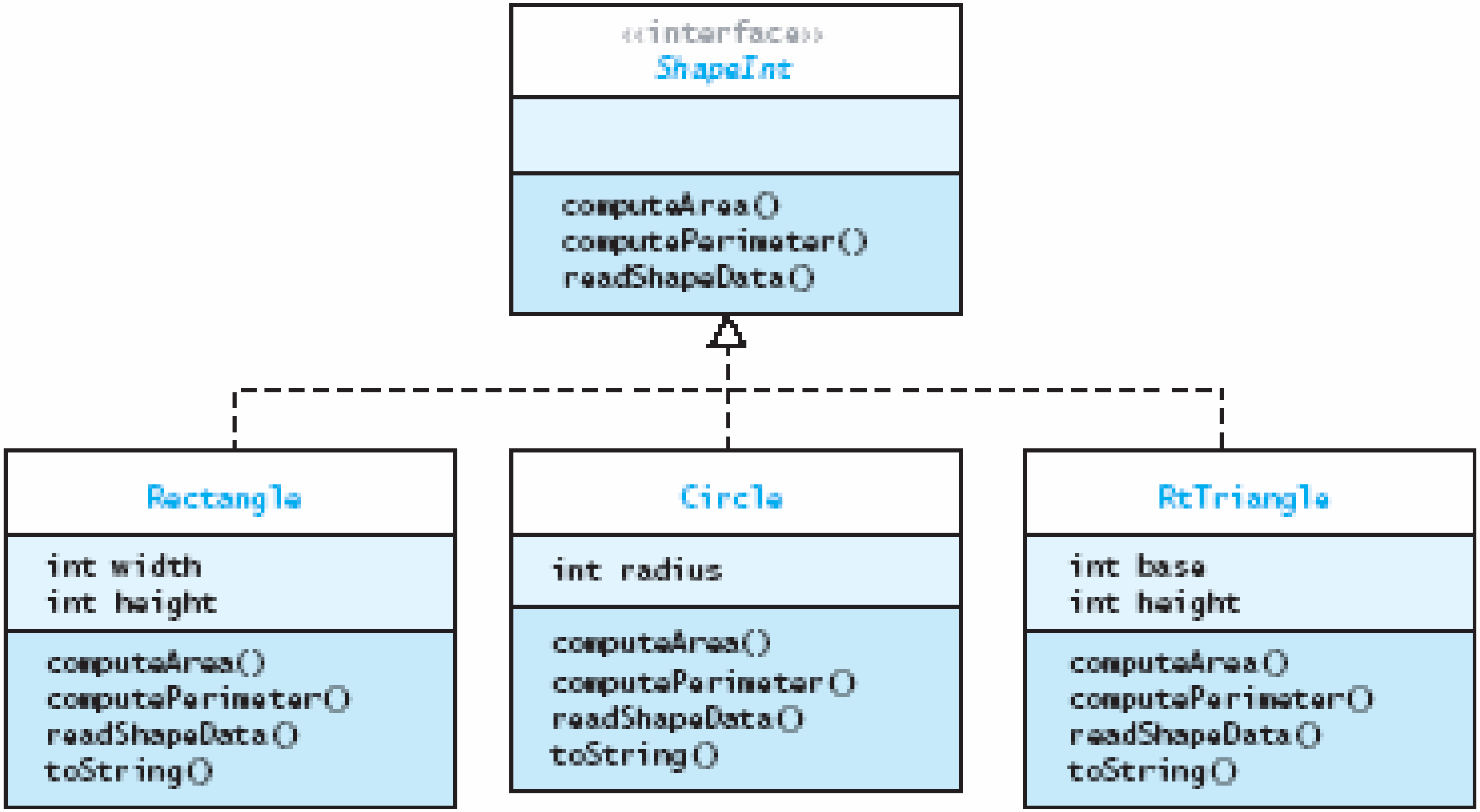
Summary of Kinds of Visibility

| Visibility | Applied to Classes | Applied to Class Members |
|---|---|---|
| `private` | Applicable to inner classes. Accessible only to members of the class in which it is declared. | Visible only within this class. |
| Default or package | Visible to classes in this package. | Visible to classes in this package. |
| `protected` | Applicable to inner classes. Visible to classes in this package and to classes outside the package that extend the class in which it is declared. | Visible to classes in this package and to classes outside the package that extend this class. |
| `public` | Visible to all classes. | Visible to all classes. The class defining the member must also be public. |

# A Shape Class Hierarchy

FIGURE 3.12

Interface ShapeInt and Three Implementors



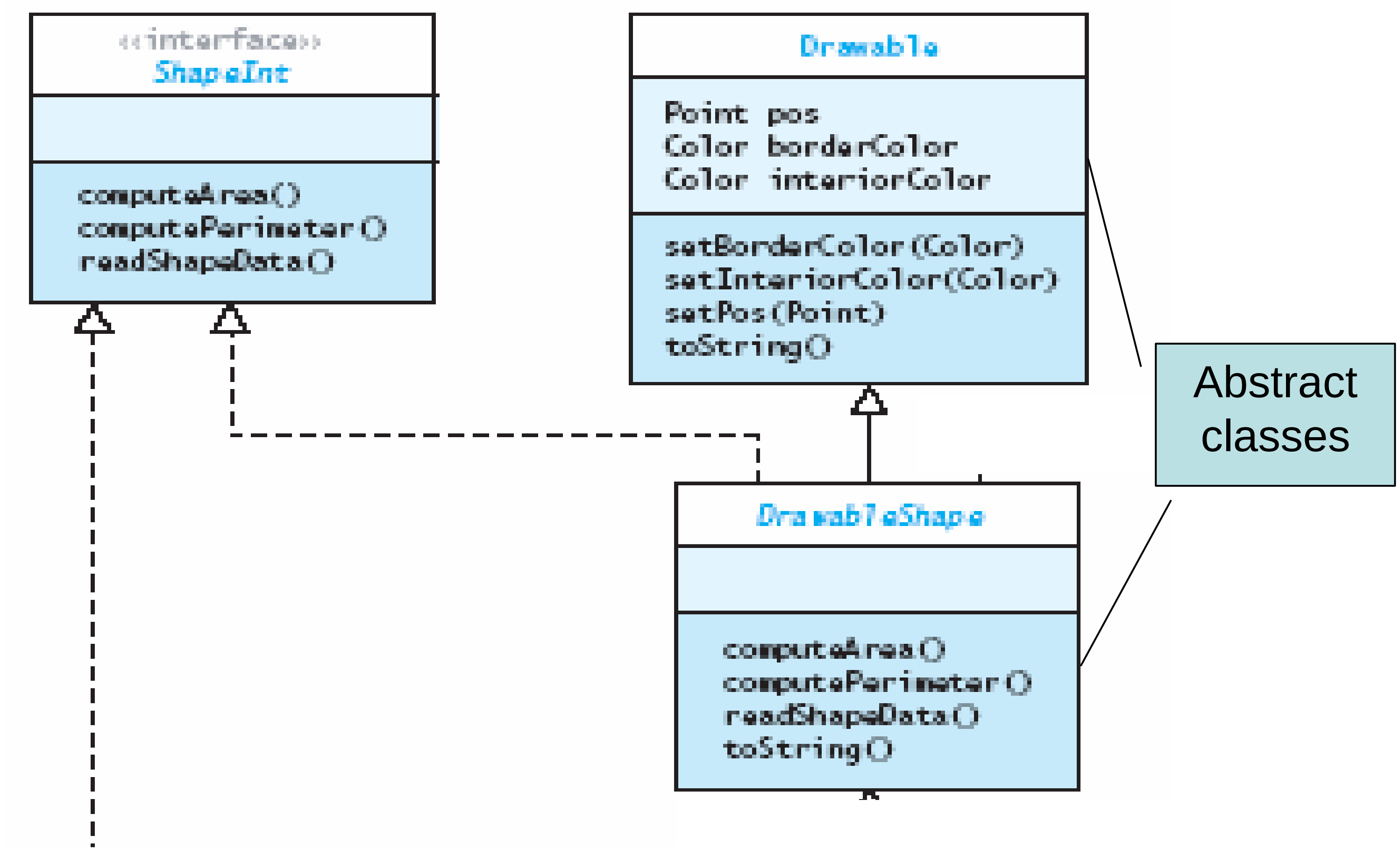
# A Shape Class Hierarchy (2)

**TABLE 3.4**
Class `Rectangle`

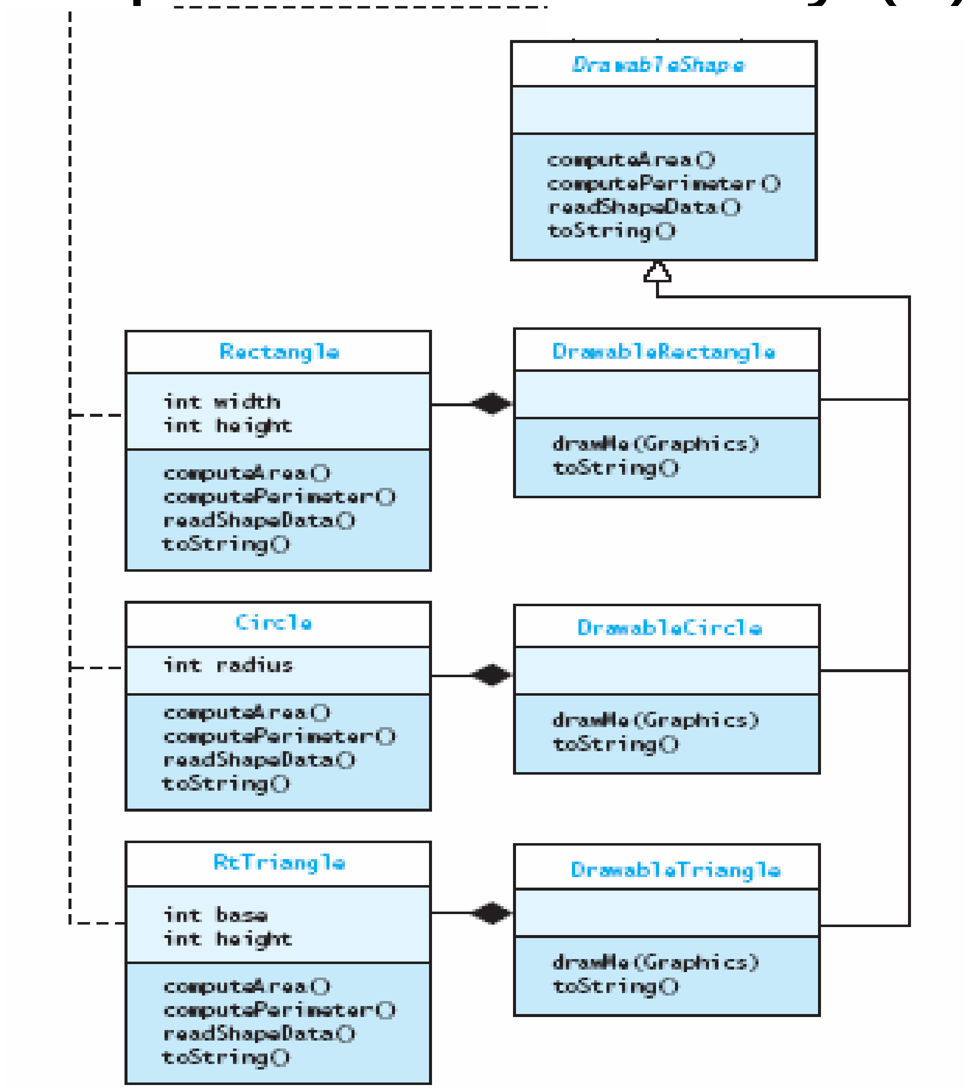
| Data Field | Attribute |
|---|---|
| `int width` | Width of a rectangle |
| `int height` | Height of a rectangle |
| **Method** | **Behavior** |
| `double computeArea()` | Computes the rectangle area (width  height). |
| `double computePerimeter()` | Computes the rectangle perimeter (2  width + 2  height). |
| `void readShapeData()` | Reads the width and height. |
| `String toString()` | Returns a string representing the state. |

# A Shape Class Hierarchy (3)

FIGURE 3.13
Drawable Shapes Hierarchy

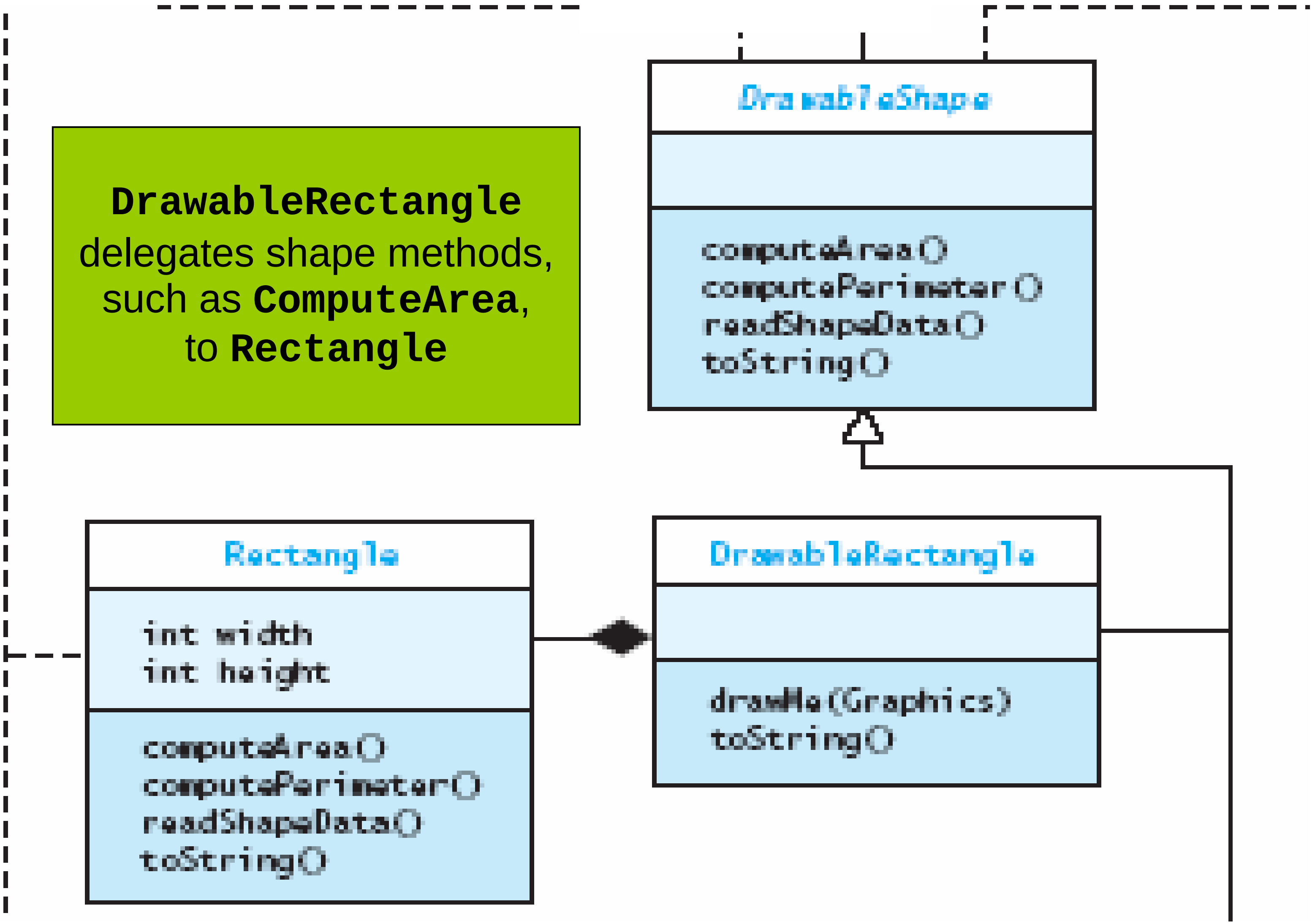


# A Shape Class Hierarchy (4)



# A Shape Class Hierarchy (5)

**`DrawableRectangle`** delegates shape methods, such as **`ComputeArea`**, to **`Rectangle`**



# A Shape Class Hierarchy (6)

**TABLE 3.5**
Class Drawable

| Data Field | Attribute |
|---|---|
| `Point pos` | (*x*, *y*) position on screen |
| `Color borderColor` | Border color |
| `Color interiorColor` | Interior color |
| **Methods** | **Behavior** |
| `void setPos(Point p)` | Sets the (*x*, *y*) screen position. |
| `void setBorderColor(Color col)` | Sets the border color to its argument. |
| `void setInteriorColor(Color col)` | Sets the interior color to its argument. |
| `String toString()` | Returns a string representing the state. |

# A Shape Class Hierarchy (7)

**TABLE 3.6**
Class DrawableShape

| Data Field | Attribute |
|---|---|
| ShapeInt theShape | Reference to an object that implements the ShapeInt interface |
| **Method** | **Behavior** |
| double computeArea() | Computes the area of the shape. |
| double computePerimeter() | Computes the perimeter of the shape. |
| void readShapeData() | Prompts for and reads the data that defines the size of the shape. |
| String toString() | Returns a string representation. |

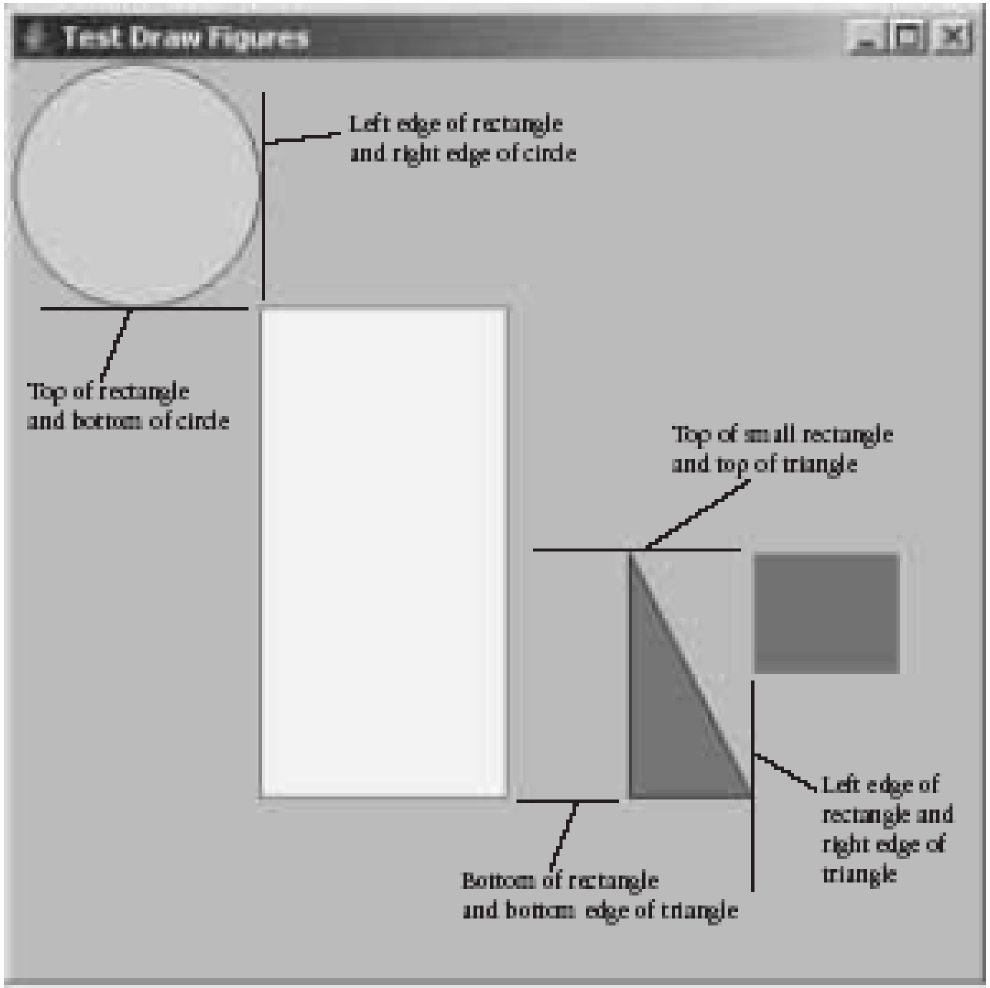
# A Shape Class Hierarchy (8)

**TABLE 3.7**
Class DrawableRectangle

| Method | Behavior |
|---|---|
| void drawMe(Graphics g) | Draws the rectangle on the screen. |
| String toString() | Returns a string representing the state. |

# A Shape Class Hierarchy (9)

FIGURE 3.15
Display of
TestDrawFigures



# Object Factories

- ***Object factory:*** method that creates instances of other classes
- Object factories are *useful when*:
  - The necessary *parameters are not known* or must be derived via computation
  - The appropriate *implementation should be selected at run time* as the result of some computation

# Example Object Factory

```
public static ShapeInt getShape () {
  String figType = JOptionPane....();
  if (figType.equalsIgnoreCase(“c”)) {
    return new Circle();
  } else if (figType.equalsIgnoreCase(“r”)) {
    return new Rectangle();
  } else if (figType.equalsIgnoreCase(“t”)) {
    return new RtTriangle();
  } else {
    return null;
  }
}
```

# Next Lecture: On to Lists!