

## Chapter 9 - Object-Oriented Programming

### Outline

- 9.1 Introduction
- 9.2 Superclasses and Subclasses
- 9.3 protected Members
- 9.4 Relationship between Superclass Objects and Subclass Objects
- 9.5 Constructors and Finalizers in Subclasses
- 9.6 Implicit Subclass-Object-to-Superclass-Object Conversion
- 9.7 Software Engineering with Inheritance
- 9.8 Composition vs. Inheritance
- 9.9 Case Study: Point, Circle, Cylinder
- 9.10 Introduction to Polymorphism
- 9.11 Type Fields and switch Statements
- 9.12 Dynamic Method Binding
- 9.13 final Methods and Classes

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

## Chapter 9 - Object-Oriented Programming

- 9.14 Abstract Superclasses and Concrete Classes
- 9.15 Polymorphism Examples
- 9.16 Case Study: A Payroll System Using Polymorphism
- 9.17 New Classes and Dynamic Binding
- 9.18 Case Study: Inheriting Interface and Implementation
- 9.19 Case Study: Creating and Using Interfaces
- 9.20 Inner Class Definitions
- 9.21 Notes on Inner Class Definitions
- 9.22 Type-Wrapper Classes for Primitive Types
- 9.23 (Optional Case Study) Thinking About Objects: Incorporating Inheritance into the Elevator Simulation
- 9.24 (Optional) Discovering Design Patterns: Introducing Creational, Structural and Behavioral Design Patterns

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

## 9.1 Introduction

- Object-oriented programming
  - Inheritance
    - Software reusability
    - Classes are created from existing ones
      - Absorbing attributes and behaviors
      - Adding new capabilities
      - **Convertible** inherits from **Automobile**
  - Polymorphism
    - Enables developers to write programs in general fashion
      - Handle variety of existing and yet-to-be-specified classes
    - Helps add new capabilities to system

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

## 9.1 Introduction (cont.)

- Object-oriented programming
  - Inheritance
    - *Subclass* inherits from *superclass*
      - Subclass usually adds instance variables and methods
    - Single vs. multiple inheritance
      - Java does not support multiple inheritance
        - Interfaces (discussed later) achieve the same effect
    - “Is a” relationship
  - Composition
    - “Has a” relationship

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

## 9.2 Superclasses and Subclasses

- “Is a” Relationship
  - Object “is an” object of another class
    - Rectangle “is a” quadrilateral
      - Class **Rectangle** inherits from class **Quadrilateral**
  - Form tree-like hierarchical structures

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

Superclass	Subclasses
Student	GraduateStudent UndergraduateStudent
Shape	Circle Triangle Rectangle
Loan	CarLoan HomeImprovementLoan MortgageLoan
Employee	FacultyMember StaffMember
Account	CheckingAccount SavingsAccount

Fig. 9.1 Some simple inheritance examples in which the subclass “is a” superclass.

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

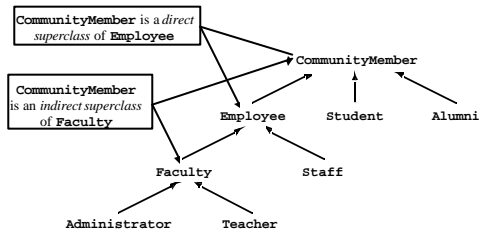
---

---

---

---

Fig. 9.2 An inheritance hierarchy for university `CommunityMembers`.



© 2002 Prentice Hall. All rights reserved.



---

---

---

---

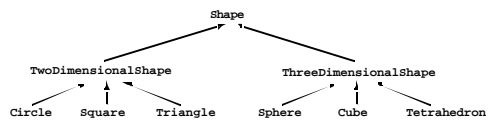
---

---

---

---

Fig. 9.3 A portion of a `Shape` class hierarchy.



© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

### 9.3 protected Members

- **protected** access members
  - Between **public** and **private** in protection
  - Accessed only by
    - Superclass methods
    - Subclass methods
    - Methods of classes in same package
      - package access

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

## 9.4 Relationship between Superclass Objects and Subclass Objects

- Subclass object
  - Can be treated as superclass object
    - Reverse is not true
      - **Shape** is not always a **Circle**
  - Every class implicitly extends **java.lang.Object**
    - Unless specified otherwise in class definition's first line

© 2002 Prentice Hall. All rights reserved.




---

---

---

---

---

---

---

---

---

---

---

---

```

1 // Fig. 9.4: Point.java
2 // Definition of class Point
3 public class Point {
4     protected int x, y; // coordinates
5
6     // No-argument constructor
7     public Point()
8     {
9         // implicit call to superclass constructor occurs here
10        setPoint( 0, 0 );
11    }
12
13    // constructor
14    public Point( int xCoordinate, int yCoordinate )
15    {
16        // implicit call to superclass constructor occurs here
17        setPoint( xCoordinate, yCoordinate );
18    }
19
20    // set x and y coordinates of Point
21    public void setPoint( int xCoordinate, int yCoordinate )
22    {
23        x = xCoordinate;
24        y = yCoordinate;
25    }
26
27    // get x coordinate
28    public int getX()
29    {
30        return x;
31    }
32 }
33
          
```

protected members prevent clients from direct access (unless clients are **Point** subclasses or are in same package)

Outline  
Point.java  
Line 5  
protected members prevent clients from direct access (unless clients are **Point** subclasses or are in same package)

© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---

---

---

---

---

```

34 // get y coordinate
35 public int getY()
36 {
37     return y;
38 }
39
40 // convert into a String representation
41 public String toString()
42 {
43     return "[" + x + ", " + y + "];"
44 }
45
46 } // end class Point
          
```

Outline  
Point.java

© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---

---

---

---

---





## 9.5 Constructors and Finalizers in Subclasses (cont.)

- **finalize** method
  - Garbage collection
  - Subclass **finalize** method
    - should invoke superclass **finalize** method

© 2002 Prentice Hall. All rights reserved.

```

1 // Fig. 9.7: Point.java
2 // Definition of class Point
3 public class Point extends Object {
4     protected int x, y; // coordinates of point
5
6     // no-argument constructor
7     public Point() {
8         x = 0;
9         y = 0;
10        System.out.println( "Point constructor: " + this );
11    }
12
13    // constructor
14    public Point( int xCoordinate, int yCoordinate )
15    {
16        x = xCoordinate;
17        y = yCoordinate;
18        System.out.println( "Point constructor: " + this );
19    }
20
21    // finaliser
22    protected void finalize() {
23        System.out.println( "Point finalizer: " + this );
24    }
25
26    // convert Point into a String representation
27    public String toString()
28    {
29        return "[" + x + ", " + y + "]";
30    }
31 } // end class Point
    
```

Point.java  
 Lines 7-20  
 Superclass constructors  
 Lines 23-26  
 Superclass **finalize** method uses **protected** for subclass access, but not for other clients

Superclass **finalize** method uses **protected** for subclass access, but not for other clients

© 2002 Prentice Hall. All rights reserved.

```

1 // Fig. 9.8: Circle.java
2 // Definition of class Circle
3 public class Circle extends Point {
4     protected double radius;
5
6     // no-argument constructor
7     public Circle() {
8         // implicit call to superclass constructor here
9         radius = 0;
10        System.out.println( "Circle constructor: " + this );
11    }
12
13    // Constructor
14    public Circle( double circleRadius, int xCoordinate, int yCoordinate )
15    {
16        // call superclass constructor
17        super( xCoordinate, yCoordinate );
18        radius = circleRadius;
19        System.out.println( "Circle constructor: " + this );
20    }
21
22    // finaliser
23    protected void finalize() {
24        System.out.println( "Circle finalizer: " + this );
25        super.finalize(); // call superclass finalize method
26    }
27 }
    
```

Circle.java  
 Line 9  
 Implicit call to **Point** constructor  
 Line 19  
 Explicit call to **Point** constructor using **super**  
 Lines 26-30  
 Override **Point**'s method **finalize**, but call it using **super**

Implicit call to **Point** constructor

Explicit call to **Point** constructor using **super**

Override **Point**'s method **finalize**, but call it using **super**

© 2002 Prentice Hall. All rights reserved.



## 9.7 Software Engineering with Inheritance

- Inheritance
  - Create class (subclass) from existing one (superclass)
    - Subclass creation does not affect superclass
  - New class inherits attributes and behaviors
  - Software reuse

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

## 9.8 Composition vs. Inheritance

- Inheritance
  - "Is a" relationship
  - **Teacher is an Employee**
- Composition
  - "Has a" relationship
  - **Employee has a TelephoneNumber**

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

## 9.9 Case Study: Point, Cylinder, Circle

- Consider point, circle, cylinder hierarchy
  - **Point** is superclass
  - **Circle** is **Point** subclass
  - **Cylinder** is **Circle** subclass

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---









```

83 // get String representation of Cylinder and calculate
84 // area and volume
85 output += "\n\nThe new location, radius " +
86 "and height of cylinder are\n" + cylinder +
87 "\narea is " + precision2.format( cylinder.area() ) +
88 "\nVolume is " + precision2.format( cylinder.volume() );
89
90 JOptionPane.showMessageDialog( null, output,
91 "Demonstrating Class Cylinder",
92 JOptionPane.INFORMATION_MESSAGE );
93
94 System.exit( 0 );
95 }
96
97 } // end class Test

```

Outline  
Test.java

© 2002 Prentice-Hall. All rights reserved.

---

---

---

---

---

---

---

---

---

---

---

---

Fig. 9.15 Testing class Test



© 2002 Prentice-Hall. All rights reserved.

---

---

---

---

---

---

---

---

---

---

---

---

### 9.10 Introduction to Polymorphism

- Polymorphism
  - Helps build extensible systems
  - Programs generically process objects as superclass objects
    - Can add classes to systems easily
      - Classes must be part of generically processed hierarchy

© 2002 Prentice-Hall. All rights reserved.

---

---

---

---

---

---

---

---

---

---

---

---

### 9.11 Type Fields and Switch Statements

- **switch**-based system
  - Determine appropriate action for object
    - Based on object's type
  - Error prone
    - Programmer can forget to make appropriate type test
    - Adding and deleting **switch** statements

© 2002 Prentice Hall. All rights reserved. 

---

---

---

---

---

---

---

---

### 9.12 Dynamic Method Binding

- Dynamic method binding
  - Implements polymorphic processing of objects
  - Use superclass reference to refer to subclass object
  - Program chooses "correct" method in subclass

© 2002 Prentice Hall. All rights reserved. 

---

---

---

---

---

---

---

---

### 9.12 Dynamic Method Binding (cont.)

- For example,
  - Superclass **Shape**
  - Subclasses **Circle**, **Rectangle** and **Square**
  - Each class draws itself according to type of class
    - **Shape** has method **draw**
    - Each subclass overrides method **draw**
    - Call method **draw** of superclass **Shape**
      - Program determines dynamically which subclass **draw** method to invoke

© 2002 Prentice Hall. All rights reserved. 

---

---

---

---

---

---

---

---

### 9.13 final Methods and Classes

- **final** method
  - Cannot be overridden in subclass
- **final** class
  - Cannot be superclass (cannot be **extended**)
    - Class cannot inherit **final** classes

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

### 9.14 Abstract Superclasses and Concrete Classes

- Abstract classes
  - Objects cannot be instantiated
  - Too generic to define real objects
    - **TwoDimensionalShape**
  - Provides superclass from which other classes may inherit
    - Normally referred to as *abstract superclasses*
- Concrete classes
  - Classes from which objects are instantiated
  - Provide specifics for instantiating objects
    - **square, Circle** and **Triangle**

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

### 9.15 Polymorphism Examples

- Video game
  - Superclass **GamePiece**
    - Contains method **drawYourself**
  - Subclasses **Martian**, **Venutian**, **LaserBeam**, etc.
    - Override method **drawYourself**
      - **Martian** draws itself with antennae
      - **LaserBeam** draws itself as bright red beam
      - This is polymorphism
  - Easily extensible
    - Suppose we add class **Mercurian**
      - Class **Mercurian** inherits superclass **GamePiece**
      - Overrides method **drawYourself**

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

## 9.16 Case Study: A Payroll System Using Polymorphism

- Abstract methods and polymorphism
  - Abstract superclass **Employee**
    - Method **earnings** applies to all employees
    - Person's earnings dependent on type of **Employee**
  - Concrete **Employee** subclasses declared **final**
    - **Boss**
    - **CommissionWorker**
    - **PieceWorker**
    - **HourlyWorker**

© 2002 Prentice Hall. All rights reserved.

```

1 // Fig. 9.16: Employee.java
2 // Abstract base class Employee.
3
4 public abstract class Employee {
5     private String firstName;
6     private String lastName;
7
8     // constructor
9     public Employee( String first, String last )
10    {
11        firstName = first;
12        lastName = last;
13    }
14
15    // get first name
16    public String getFirstName()
17    {
18        return firstName;
19    }
20
21    // get last name
22    public String getLastName()
23    {
24        return lastName;
25    }
26
27    public String toString()
28    {
29        return firstName + ' ' + lastName;
30    }
31

```

Annotations in the image:

- Line 4: **abstract class cannot be instantiated**
- Line 4: **abstract class can have instance data**
- Line 4: **abstract class can have constructors for subclasses to initialize inherited data**
- Line 9-13: **abstract class can have instance data and non-abstract methods for subclasses**
- Line 16-20: **abstract class can have constructors for subclasses to initialize inherited data**
- Line 22-25: **abstract class can have instance data and non-abstract methods for subclasses**
- Line 27-30: **abstract class can have constructors for subclasses to initialize inherited data**

Outline:

- Employee.java
- Lines 9-13: abstract class can have instance data and non-abstract methods for subclasses
- Lines 16-20: abstract class can have constructors for subclasses to initialize inherited data

© 2002 Prentice Hall. All rights reserved.

```

32 // Abstract method that must be implemented for each
33 // derived class of Employee from which objects
34 // are instantiated.
35 public abstract double earnings();
36
37 } // end class Employee

```

Annotations in the image:

- Line 35: **Subclasses must implement abstract method**

Outline:

- Employee.java
- Line 35: Subclasses must implement abstract method

© 2002 Prentice Hall. All rights reserved.


















 Outline  
 Test.java



© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---


---

---

---

### 9.19 Case Study: Creating and Using Interfaces

- Use **interface Shape**
  - Replace **abstract class Shape**
- **Interface**
  - Definition begins with **interface** keyword
  - Classes **implement** an interface (and its methods)
  - Contains **public abstract** methods
    - Classes (that **implement** the interface) must implement these methods



© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---



---

---

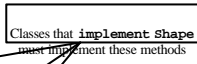
---

```

1 // Fig. 9.27: Shape.java
2 // Definition of interface Shape
3
4 public interface Shape {
5
6     // calculate area
7     public abstract double area(); ▲
8
9     // calculate volume
10    public abstract double volume(); ▲
11
12    // return shape name
13    public abstract String getname();
14 }
          
```

 Outline  
 Shape.java  
 Lines 7-13  
 Classes that  
**implement Shape**  
 must implement these  
 methods

Classes that **implement Shape** must implement these methods



© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---

---

---







## 9.20 Inner Class Definitions

- Inner classes
  - Class is defined inside another class body
  - Frequently used with GUI handling
    - Declare **ActionListener** inner class
    - GUI components can register **ActionListeners** for events
      - Button events, key events, etc.

© 2002 Prentice Hall. All rights reserved.




---

---

---

---

---

---

---

---

---

---

---

---

```

1 // Fig. 9.32: Time.java
2 // Time class definition.
3
4 // Java core packages
5 import java.text.DecimalFormat;
6
7 // This class maintains the time in 24-hour format
8 public class Time extends Object {
9     private int hour; // 0 - 23
10    private int minute; // 0 - 59
11    private int second; // 0 - 59
12
13    // Time constructor initializes each instance variable
14    // to zero. Ensures that Time object starts in a
15    // consistent state.
16    public Time()
17    {
18        setTime( 0, 0, 0 );
19    }
20
21    // Set a new time value using universal time. Perform
22    // validity checks on the data. Set invalid values to zero.
23    public void setTime( int hour, int minute, int second )
24    {
25        setHour( hour );
26        setMinute( minute );
27        setSecond( second );
28    }
29
30    // validate and set hour
31    public void setHour( int h )
32    {
33        hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
34    }
35

```

Outline  
Time.java  
Line 8  
Same Time class used in Chapter 8

Same Time class used in Chapter 8

© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---

---

---

---

---

```

36 // validate and set minute
37 public void setMinute( int m )
38 {
39     minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
40 }
41
42 // validate and set second
43 public void setSecond( int s )
44 {
45     second = ( ( s >= 0 && s < 60 ) ? s : 0 );
46 }
47
48 // get hour
49 public int getHour()
50 {
51     return hour;
52 }
53
54 // get minute
55 public int getMinute()
56 {
57     return minute;
58 }
59
60 // get second
61 public int getSecond()
62 {
63     return second;
64 }
65

```

Outline  
Time.java  
Mutator and accessor methods

Mutator and accessor methods

© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---

---

---

---

---





## 9.20 Inner Class Definitions (cont.)

- Anonymous inner class
  - Inner class without name
  - Created when class is defined in program

© 2002 Prentice Hall. All rights reserved.

```

1 // Fig. 9.34: TimeTestWindow.java
2 // Demonstrating the Time class set and get methods
3
4 // Java core packages
5 import java.awt.*;
6 import java.awt.event.*;
7
8 // Java extension packages
9 import javax.swing.*;
10
11 public class TimeTestWindow extends JFrame {
12     private Time time;
13     private JLabel hourLabel, minuteLabel, secondLabel;
14     private JTextField hourField, minuteField,
15         secondField, displayField;
16
17     // set up GUI
18     public TimeTestWindow()
19     {
20         super( "Inner Class Demonstration" );
21
22         // create Time object
23         time = new Time();
24
25         // create GUI
26         Container container = getContentPane();
27         container.setLayout( new FlowLayout() );
28
29         hourLabel = new JLabel( "Set Hour" );
30         hourField = new JTextField( 10 );
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61

```

Outline  
TimeTestWindow.j  
ava

© 2002 Prentice Hall.  
All rights reserved.

```

32 // register hourField event handler
33 hourField.addActionListener(
34     // anonymous inner class
35     new ActionListener() {
36         public void actionPerformed( ActionEvent event )
37         {
38             time.setHour(
39                 Integer.parseInt( event.getActionCommand() ) );
40             hourField.setText( "" );
41             displayTime();
42         }
43     } // end anonymous inner class
44 ); // end call to addActionListener
45
46 container.add( hourLabel );
47 container.add( hourField );
48
49 minuteLabel = new JLabel( "Set minute" );
50 minuteField = new JTextField( 10 );
51
52 // register minuteField event handler
53 minuteField.addActionListener(
54     // anonymous inner class
55     new ActionListener() {
56         public void actionPerformed( ActionEvent event )
57         {
58             time.setMinute(
59                 Integer.parseInt( event.getActionCommand() ) );
60             minuteField.setText( "" );
61             displayTime();
62         }
63     } // end anonymous inner class
64 ); // end call to addActionListener
65
66 container.add( minuteLabel );
67 container.add( minuteField );
68
69 secondLabel = new JLabel( "Set second" );
70 secondField = new JTextField( 10 );
71
72 // register secondField event handler
73 secondField.addActionListener(
74     // anonymous inner class
75     new ActionListener() {
76         public void actionPerformed( ActionEvent event )
77         {
78             time.setSecond(
79                 Integer.parseInt( event.getActionCommand() ) );
80             secondField.setText( "" );
81             displayTime();
82         }
83     } // end anonymous inner class
84 ); // end call to addActionListener
85
86 container.add( secondLabel );
87 container.add( secondField );
88
89 displayField = new JTextField( 10 );
90 container.add( displayField );
91
92 container.setVisible( true );
93
94 setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
95
96 TimeTestWindow ttw = new TimeTestWindow();
97 ttw.pack();
98 ttw.setVisible( true );
99 ttw.wait();
100
101 } // end TimeTestWindow

```

Outline  
TimeTestWindow.j  
ava

© 2002 Prentice Hall.  
All rights reserved.

```

82     public void actionPerformed( ActionEvent event )
83     {
84         time.setMinute(
85             Integer.parseInt( event.getActionCommand() ) );
86         minuteField.setText( "" );
87         displayTime();
88     }
89     } // end anonymous inner class
90 } // end call to addActionListener
91
92 container.add( minuteLabel );
93 container.add( minuteField );
94
95 secondLabel = new JLabel( "Set Second" );
96 secondField = new JTextField( 10 );
97
98 secondField.addActionListener(
99     // anonymous inner class
100     new ActionListener() {
101         public void actionPerformed( ActionEvent event )
102         {
103             time.setSecond(
104                 Integer.parseInt( event.getActionCommand() ) );
105             secondField.setText( "" );
106             displayTime();
107         }
108     } // end anonymous inner class
109 ); // end call to addActionListener
110

```

Outline  
TimeTestWindow.java  
Line 64-67  
Logic differs from logic in actionPerformed method of hourField's inner class  
Repeat process for JTextField secondField  
Line 87-90  
Logic differs from logic in actionPerformed methods of other inner classes  
© 2002 Prentice Hall. All rights reserved.

```

107 container.add( secondLabel );
108 container.add( secondField );
109 displayField = new JTextField( 30 );
110 displayField.setEditable( false );
111 container.add( displayField );
112 }
113
114 // display time in displayField
115 public void displayTime()
116 {
117     displayField.setText( "The time is: " + time );
118 }
119
120 // create TimeTestWindow, register for its window events
121 // and display it to begin application's execution
122 public static void main( String args[] )
123 {
124     TimeTestWindow window = new TimeTestWindow();
125     // register listener for windowClosing event
126     window.addWindowListener(
127         // anonymous inner class for windowClosing event
128         new WindowAdapter() {
129             // terminate application when user closes window
130             public void windowClosing( WindowEvent event )
131             {
132                 System.exit( 0 );
133             }
134         } // end anonymous inner class
135     ); // end call to addWindowListener
136 }

```

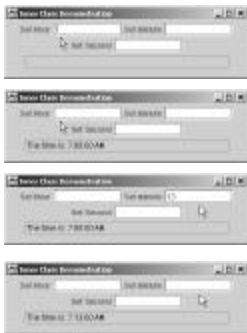
Outline  
TimeTestWindow.java  
Line 118-131  
Define anonymous inner class that extends WindowAdapter to enable closing of JFrame  
Define anonymous inner class that extends WindowAdapter to enable closing of JFrame  
© 2002 Prentice Hall. All rights reserved.



```


132 window.setSize( 400, 120 );
133 window.setVisible( true );
134 }
135 } // end class TimeTestWindow

```

Outline  
TimeTestWindow.java  
ava  
© 2002 Prentice Hall. All rights reserved.



 **Outline**  
 **TimeTestWindow.java**



© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---

### 9.21 Notes on Inner Class Definitions

- Notes for inner-class definition and use
  - Compiling class that contains inner class
    - Results in separate **.class** file
  - Inner classes with class names
    - **public, protected, private** or package access
  - Access outer class' **this** reference
    - *OuterClassName.this*
  - Outer class is responsible for creating inner class objects
  - Inner classes can be declared **static**

© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---

### 9.22 Type-Wrapper Classes for Primitive Types

- Type-wrapper class
  - Each primitive type has one
    - **Character, Byte, Integer, Boolean**, etc.
  - Enable to represent primitive as **Object**
    - Primitive data types can be processed polymorphically
  - Declared as **final**
  - Many methods are declared **static**

© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---

### 9.23 (Optional Case Study) Thinking About Objects: Incorporating Inheritance into the Elevator Simulation

- Our design can benefit from inheritance
  - Examine sets of classes
  - Look for commonality between/among sets
    - Extract commonality into superclass
      - Subclasses inherits this commonality

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

### 9.23 Thinking About Objects (cont.)

- **ElevatorButton** and **FloorButton**
  - Treated as separate classes
  - Both have *attribute* **pressed**
  - Both have *behaviors* **pressButton** and **resetButton**
  - Move attribute and behaviors into superclass **Button**?
    - We must examine whether these objects have distinct behavior
      - If same behavior
        - They are objects of class **Button**
      - If different behavior
        - They are objects of distinct **Button** subclasses

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

Fig. 9.35 Attributes and operations of classes **FloorButton** and **ElevatorButton**.



© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

### 9.23 Thinking About Objects (cont.)

- **ElevatorButton** and **FloorButton**
  - **FloorButton** requests **Elevator** to **Floor** of request
    - **Elevator** will sometimes respond
  - **ElevatorButton** signals **Elevator** to move
    - **Elevator** will always respond
  - Neither button *decides* for the **Elevator** to move
    - **Elevator** decides itself
  - Both buttons signal **Elevator** to move
    - Therefore, both buttons exhibit identical behavior
      - They are objects of class **Button**
      - Combine (not inherit) **ElevatorButton** and **FloorButton** into class **Button**

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

### 9.23 Thinking About Objects (cont.)

- **ElevatorDoor** and **FloorDoor**
  - Treated as separate classes
  - Both have *attribute* **open**
  - Both have *behaviors* **openDoor** and **closeDoor**
  - Both door "inform" a **Person** that a door has opened
    - both doors exhibit identical behavior
      - They are objects of class **Door**
      - Combine (not inherit) **ElevatorDoor** and **FloorDoor** into class **Door**

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

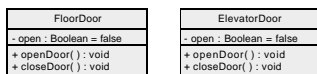
---

---

---

---

Fig. 9.36 Attributes and operations of classes **FloorDoor** and **ElevatorDoor**



© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

### 9.23 Thinking About Objects (cont.)

- Representing location of **Person**
  - On what **Floor** is **Person** when riding **Elevator**?
  - Both **Floor** and **Elevator** are types of locations
    - Share **int** attribute **capacity**
    - Inherit from **abstract** superclass **Location**
      - Contains **String** **locationName** representing location
        - "firstFloor"
        - "secondFloor"
        - "elevator"
  - **Person** now contains **Location** reference
    - References **Elevator** when person is in elevator
    - References **Floor** when person is on floor

© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---

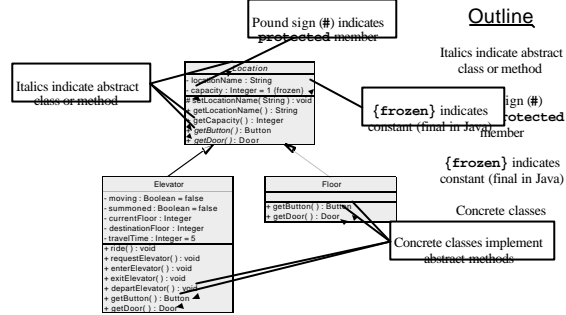
---

---

---

---

Fig. 9.37 Class diagram modeling generalization of superclass **Location** and subclasses **Elevator** and **Floor**.



© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---

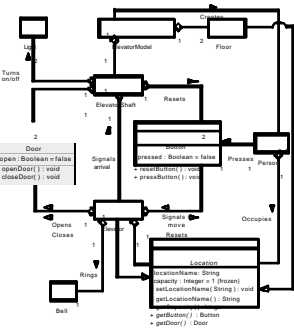
---

---

---

---

Fig. 9.38 Class diagram of our simulator (incorporating inheritance).



© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---

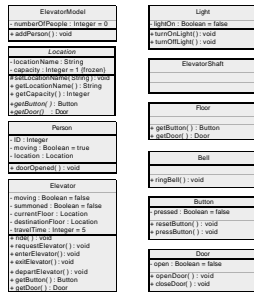
---

---

---

---

Fig. 9.39 Class diagram with attributes and operations (incorporating inheritance).



© 2002 Prentice Hall. All rights reserved.

### 9.23 Thinking About Objects (cont.)

- Continue implementation
  - Transform design (i.e., class diagram) to code
  - Generate “skeleton code” with our design
    - Use class **Elevator** as example
    - Two steps (incorporating inheritance)

© 2002 Prentice Hall. All rights reserved.

### 9.23 Thinking About Objects (cont.) Step 1

```

public class Elevator extends Location {

    // class constructor
    public Elevator() {}



}
    
```

© 2002 Prentice Hall. All rights reserved.

```

1 // Elevator.java
2 // Generated using class diagrams 9.38 and 9.39
3 public class Elevator extends Location {
4
5     // class attributes
6     private boolean moving;
7     private boolean summoned;
8     private Location currentFloor;
9     private Location destinationFloor;
10    private int travelTime = 5;
11    private Button elevatorButton;
12    private Door elevatorDoor;
13    private Bell bell;
14
15    // class constructor
16    public Elevator() {}
17
18    // class methods
19    public void ride() {}
20    public void requestElevator() {}
21    public void enterElevator() {}
22    public void exitElevator() {}
23    public void departElevator() {}
24
25    // method overriding getButton
26    public Button getButton() {
27        {
28            return elevatorButton;
29        }
30    }
31
32    // method overriding getDoor
33    public Door getDoor() {
34        {
35            return elevatorDoor;
36        }
37    }

```

 Outline  
 Step 2  
 Implement abstract classes

Implement abstract classes

© 2002 Prentice-Hall. All rights reserved.

---

---

---

---

---

---

---

---

---


---

---

---

### 9.24 (Optional) Discovering Design Patterns: Introducing Creational, Structural and Behavioral Design Patterns

- Design-patterns discussion
  - Discuss each type
    - Creational
    - Structural
    - Behavioral
  - Discuss importance
  - Discuss how we can use each pattern in Java



© 2002 Prentice-Hall. All rights reserved.

---

---

---

---

---

---

---

---

---


---

---

---

### 9.24 Discovering Design Patterns (cont.)

- We also introduce
  - *Concurrent* design patterns
    - Used in multithreaded systems
    - Chapter 15
  - *Architectural* patterns
    - Specify how subsystems interact with each other
    - Chapter 17



© 2002 Prentice-Hall. All rights reserved.

---

---

---

---

---

---

---

---

---

---

---

---



## 9.24 Discovering Design Patterns (cont.)

- 5 creational design patterns
  - Abstract Factory (Chapter 17)
  - Builder (not discussed)
  - Factory Method (Chapter 13)
  - Prototype (Chapter 21)
  - Singleton (Chapter 9)

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

## 9.24 Discovering Design Patterns (cont.)

- Singleton
  - Used when system should contain *exactly* one object of class
    - e.g., one object manages database connections
  - Ensures system instantiates *maximum* of one class object

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

```
1 // Singleton.java
2 // Demonstrates Singleton design pattern
3 package com.deitel.jhtp4.designpatterns;
4
5 public final class Singleton {
6
7     // Singleton object returned by method getSingleton()
8     private static Singleton singleton;
9
10    // constructor prevents instantiation from outside class
11    private Singleton() {
12        System.err.println( "Singleton object created." );
13    }
14
15    // create Singleton and ensure only one Singleton instance
16    public static Singleton getSingletonInstance()
17    {
18        // instantiate Singleton if null
19        if ( singleton == null )
20            singleton = new Singleton();
21    }
22    return singleton;
23 }
24
25 }
```

**Outline**

Singleton.java

private constructor ensures only class Singleton can instantiate Singleton object

only class Singleton can instantiate Singleton object

Lines 20-23 Instantiate Singleton object only once, but return same reference

Instantiate Singleton object only once, but return same reference

© 2002 Prentice Hall. All rights reserved.

---

---

---

---

---

---

---

---



### 9.24 Discovering Design Patterns (cont.)

- Behavioral design patterns
  - Model how objects collaborate with one another
  - Assign responsibilities to algorithms

© 2002 Prentice Hall. All rights reserved. 

---

---

---

---

---

---

---

---

### 9.24 Discovering Design Patterns (cont.)

- Behavioral design patterns
  - Chain-of-Responsibility (Chapter 13)
  - Command (Chapter 13)
  - Interpreter (not discussed)
  - Iterator (Chapter 21)
  - Mediator (not discussed)
  - Memento (Chapter 9)
  - Observer (Chapter 13)
  - State (Chapter 9)
  - Strategy (Chapter 13)
  - Template Method (Chapter 13)
  - Visitor (not discussed)

© 2002 Prentice Hall. All rights reserved. 

---

---

---

---

---

---

---

---

### 9.24 Discovering Design Patterns (cont.)

- Memento
  - Allows object to save its *state* (set of attribute values)
  - Consider painting program for creating graphics
    - Offer “undo” feature if user makes mistake
      - Returns program to previous state (before error)
    - *History* lists previous program states
  - *Originator object* occupies state
    - e.g., drawing area
  - *Memento object* stores copy of originator object’s attributes
    - e.g., memento saves state of drawing area
  - *Caretaker object* (history) contains references to mementos
    - e.g., history lists mementos from which user can select

© 2002 Prentice Hall. All rights reserved. 

---

---

---

---

---

---

---

---

### 9.24 Discovering Design Patterns (cont.)

- State
  - Encapsulates object's state
  - Consider optional elevator-simulation case study
    - Person walks on floor toward elevator
      - Use integer to represent floor on which person walks
    - Person rides elevator to other floor
    - On what floor is the person when riding elevator?

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---

### 9.24 Discovering Design Patterns (cont.)

- State
  - We implement a solution:
    - Abstract superclass **Location**
    - Classes **Floor** and **Elevator** extend **Location**
    - Encapsulates information about person location
      - Each location has reference to **Button** and **Door**
    - Class **Person** contains **Location** reference
      - Reference **Floor** when on floor
      - Reference **Elevator** when in elevator

© 2002 Prentice Hall. All rights reserved.



---

---

---

---

---

---

---

---