Chapter 8 – Object-Based Programming

Outline

8.1 Introduction
8.2 Implementing a Time Abstract Data Type with a Class
8.3 Class Scope
8.4 Controlling Access to Members
8.5 Referring to the Current Object's Members with `this`
8.6 Initializing Class Objects: Constructors
8.7 Using Overloaded Constructors
8.8 Using Set and Get Methods
8.9 Composition
8.10 Garbage Collection
8.11 Static Class Members
8.12 Final Instance Variables
8.13 Creating Packages
8.14 Package Access
8.15 Software Reusability
8.16 Data Abstraction and Encapsulation
8.17 (Optional Case Study) Thinking About Objects: Starting to Program the Classes for the Elevator Simulation

8.1 Introduction

- Object Oriented Programming (OOP)
  - *Encapsulates* data (attributes) and methods (behaviors)
    - Objects
  - Allows objects to communicate
    - Well-defined *interfaces*

8.1 Introduction (cont.)

- Procedural programming language
  - C is an example
  - Action-oriented
  - Functions are units of programming
- Object-oriented programming language
  - Java is an example
  - Object-oriented
  - Classes are units of programming
    - Functions, or *methods*, are encapsulated in classes

8.1 Introduction (cont.)

- This chapter discusses
  - How to create objects
  - How to use objects

◀ ▶

8.2 Implementing a Time Abstract Data Type with a Class

- We introduce classes `Time1` and `TimeTest`
  - `Time1.java` declares class `Time1`
  - `TimeTest.java` declares class `TimeTest`
  - `public` classes must be declared in separate files
  - Class `Time1` will not execute by itself
    - Does not have method `main`
    - `TimeTest`, which has method `main`, creates (*instantiates*) and uses `Time1` object

Outline

```
1     // Fig. 8.1: Time1.java
2     // Time1 class declaration maintains the time in 24-hour format.
3     import java.text.DecimalFormat;
4
5     public class Time1 extends Object {
6        private int hour;     // 0 - 23
7        private int minute;   // 0 - 59
8        private int second;   // 0 - 59
9
10       // Time1 constructor initializes each instance variable to zero;
11       // ensures that each Time1 object starts in a consistent state
12       public Time1()
13       {
14          setTime( 0, 0, 0 );
15       }
16
17       // set a new time value using universal tim
18       // validity checks on the data; set invalid
19       public void setTime( int h, int m, int s )
20       {
21          hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
22          minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
23          second = ( ( s >= 0 && s < 60 ) ? s : 0 );
24       }
25
```

`Time1` (subclass) extends superclass `java.lang.Object` (Chapter 9 discusses inheritance)

`private` variables (and methods) are accessible only to methods in this class

`Time1` constructor creates `Time1` object then invokes method `setTime`

Meth… `setTime` sets `private` vari…

`public` methods (and variables) are accessible wherever program has `Time1` reference

`Time1.java`

Line 5

…ss)

…ass

`java.lang.Objec`
`t`

Lines 6-8

`private` variables

Lines 12-15

`Time1` constructor then invokes method `setTime`

Line 19

…`ic` methods

…`Time`

…`e`

variables according to arguments

```java
   // convert to String in universal-time format
   public String toUniversalString()
   {
      DecimalFormat twoDigits = new DecimalFormat( "00" );

      return twoDigits.format( hour ) + ":" +
         twoDigits.format( minute ) + ":" + twoDigits.format( second );
   }

   // convert to String in standard-time format
   public String toStandardString()
   {
      DecimalFormat twoDigits = new DecimalFormat( "00" );

      return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) + ":" +
         twoDigits.format( minute ) + ":" + twoDigits.format( second ) +
         ( hour < 12 ? " AM" : " PM" );
   }

} // end class Time1
```

Outline

Time1.java

8.2 Implementing a Time Abstract Data Type with a Class (cont.)

- Every Java class must extend another class
  - `Time1` extends `java.lang.Object`
  - If class does not explicitly `extend` another class
    - class implicitly extends `Object`
- Class *constructor*
  - Same name as class
  - Initializes instance variables of a class object
  - Called when program instantiates an object of that class
  - Can take arguments, but *cannot return values*
  - Class can have several constructors, through *overloading*
  - Class `Time1` constructor(lines 12-15)

Outline

```java
1       // Fig. 8.2: TimeTest1.java
2       // Class TimeTest1 to exercise class
3       import javax.swing.JOptionPane;
4
5       public class TimeTest1 {
6
7          public static void main( String args[] )
8          {
9             Time1 time = new Time1();  // calls Time1
10
11            // append String version of time to String output
12            String output = "The initial universal time is: " +
13               time.toUniversalString() + "\nThe initial standard time is: " +
14               time.toStandardString();
15
16            // change time and append updated time to output
17            time.setTime( 13, 27, 6 );
18            output += "\n\nUniversal time after setTime is: " +
19               time.toUniversalString() +
20               "\nStandard time after setTime is: " + time.toStandardString();
21
22            // set time with invalid values; append updated time to output
23            time.setTime( 99, 99, 99 );
24            output += "\n\nAfter attempting invalid settings: " +
25               "\nUniversal time: " + time.toUniversalString() +
26               "\nStandard time: " + time.toStandardString();
27
```

Declare and create instance of class `Time1` by calling `Time1` constructor

`TimeTest1` interacts with `Time1` by calling `Time1` `public` methods

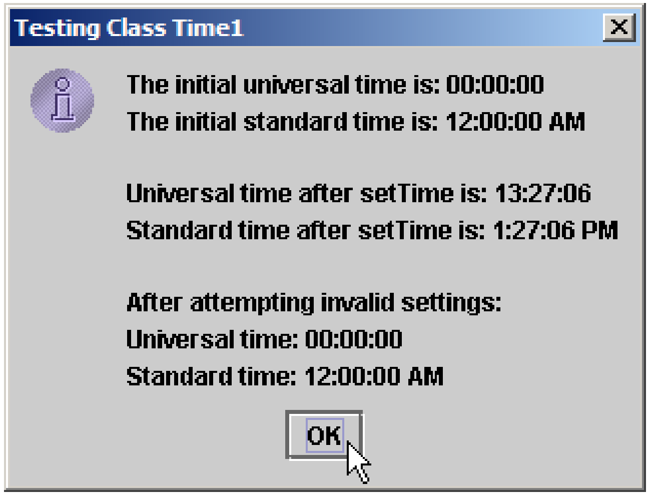
`TimeTest1.java`

Line 9
Declare and create
e of class
by calling
`Time1` constructor

Lines 12-26
`TimeTest1` interacts with `Time1` by calling `Time1` `public` methods

TimeTest1.java

```
      JOptionPane.showMessageDialog( null, output,
         "Testing Class Time1", JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );

   } // end main

} // end class TimeTest1
```



```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

8.3 Class Scope

- Class scope
  - Class variables and methods
  - Members are accessible to all class methods
  - Members can be referenced by name
    - objectReferenceName.objectMemberName
  - Shadowed (hidden) class variables
    - `this`.variableName

8.4 Controlling Access to Members

- Member access modifiers
  - Control access to class's variables and methods
  - `public`
    - Variables and methods accessible to clients of the class
  - `private`
    - Variables and methods not accessible to clients of the class

```
1     // Fig. 8.3: TimeTest2.java
2     // Errors resulting from attempts to access private members of Time1.
3     public class TimeTest2 {
4
5        public static void main( String args[] )
6        {
7           Time1 time = new Time1();
8
9           time.hour = 7;    // error: hour is a private instance variable
10          time.minute = 15; // error: minute is a private instance variable
11          time.second = 30; // error: second is a private instance variable
12       }
13
14    } // end class TimeTest2
```

Compiler error – TimeTest2 cannot directly access Time1's private data

```
TimeTest2.java:9: hour has private access in Time1
      time.hour = 7;    // error: hour is a private instance variable
          ^
TimeTest2.java:10: minute has private access in Time1
      time.minute = 15; // error: minute is a private instance variable
          ^
TimeTest2.java:11: second has private access in Time1
      time.second = 30; // error: second is a private instance variable
          ^
3 errors
```

TimeTest2.java

Lines 9-11

Compiler error – TimeTest2 cannot directly access Time1's private data

8.5 Referring to the Current Object's Members with `this`

- Keyword `this` (*`this` reference*)
  - Allows an object to refers to itself

```java
// Fig. 8.4: ThisTest.java
// Using the this reference to refer to instance variables and methods.
import javax.swing.*;
import java.text.DecimalFormat;

public class ThisTest {

   public static void main( String args[] )
   {
      SimpleTime time = new SimpleTime( 12, 30, 19 );

      JOptionPane.showMessageDialog( null, time.buildString(),
         "Demonstrating the \"this\" Reference",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   }

} // end class ThisTest

// class SimpleTime demonstrates the "this" reference
class SimpleTime {
   private int hour;
   private int minute;
   private int second;

```

Outline

ThisTest.java

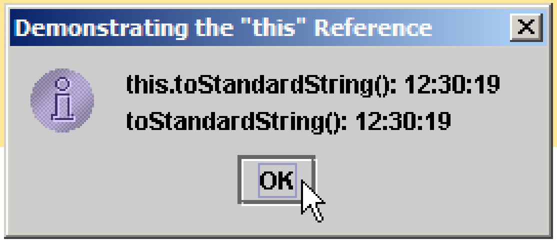
Outline

ThisTest.java

```
27       // constructor uses parameter names identical to instance variable
28       // names; "this" reference required to distinguish between names
29       public SimpleTime( int hour, int minute, int second )
30       {
31          this.hour = hour;       // set "this" object'
32          this.minute = minute;   // set "this" object'
33          this.second = second;   // set "this" object'
34       }
35
36       // use explicit and implicit "this" to call toStandardString
37       public String buildString()
38       {
39          return "this.toStandardString(): " + this.toStandardString() +
40             "\ntoStandardString(): " + toStandardString();
41       }
42
43       // return String representation of SimpleTime
44       public String toStandardString()
45       {
46          DecimalFormat twoDigits = new DecimalFormat( "00" );
47
48          // "this" is not required here, because method does not
49          // have local variables with same names as instance variables
50          return twoDigits.format( this.hour ) + ":" +
51             twoDigits.format( this.minute ) + ":" +
52             twoDigits.format( this.second );
53       }
54
55    } // end class SimpleTime
```

`this` used to distinguish between arguments and `ThisTest` variables

Use explicit and implicit this to call toStandardString



```
this.toStandardString(): 12:30:19
toStandardString(): 12:30:19
```

Lines 31-33
this used to distinguish between argumens and variables

Lines 39-40
use explicit and
o call
ng

8.6 Initializing Class Objects: Constructors

- Class constructor
  - Same name as class
  - Initializes instance variables of a class object
  - Call class constructor to instantiate object of that class

    `new` *ClassName*`(` *argument1*, *argument2, …, arugmentN* `);`

    - `new` indicates that new object is created
    - *ClassName* indicates type of object created
    - *arguments* specifies constructor argument values

8.7 Using Overloaded Constructors

- Overloaded constructors
  - Methods (in same class) may have same name
  - Must have different parameter lists

```
1       // Fig. 8.5: Time2.java
2       // Time2 class declaration with overloaded constructors.
3       import java.text.DecimalFormat;
4
5       public class Time2 {
6          private int hour;     // 0 - 23
7          private int minute;   // 0 - 59
8          private int second;   // 0 - 59
9
10        // Time2 constructor initializes each
11        // ensures that Time object starts in a consistent state
12        public Time2()
13        {
14           this( 0, 0, 0 ); // invoke Time2 constr                        s
15        }
16
17        // Time2 constructor: hour supplied, minute and second defaulted to 0
18        public Time2( int h )
19        {
20           this( h, 0, 0 ); // invoke Time2 con
21        }
22
23        // Time2 constructor: hour and minute supplied, second defaulted to 0
24        public Time2( int h, int m )
25        {
26           this( h, m, 0 ); // invoke Time2 constructor with three arguments
27        }
28
```

No-argument (default) constructor

Use this to invoke the Time2 constructor declared at lines 30-33

Overloaded constructor has one `int` argument

Second overloaded constructor has two `int` arguments

`Time2.java`

Lines 12-15
o-argument (default)
onstructor
Line 14
Use this to invoke the Time2 constructor declared at lines 30-33
Lines 18-21
Overloaded constructor has one `int` argument
Lines 24-27
Second overloaded constructor has two `int` arguments

```
29    // Time2 constructor: hour, minute and second supplied
30    public Time2( int h, int m, int s )
31    {
32       setTime( h, m, s ); // invoke setTime to validate time
33    }
34
35    // Time2 constructor: another Time2 object supplied
36    public Time2( Time2 time )
37    {
38       // invoke Time2 constructor with three arguments
39       this( time.hour, time.minute, time.second );
40    }
41
42    // set a new time value using universal time; perform
43    // validity checks on data; set invalid values to zero
44    public void setTime( int h, int m, int s )
45    {
46       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
47       minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
48       second = ( ( s >= 0 && s < 60 ) ? s : 0 );
49    }
50
51    // convert to String in universal-time format
52    public String toUniversalString()
53    {
54       DecimalFormat twoDigits = new DecimalFormat( "00" );
55
56       return twoDigits.format( hour ) + ":" +
57          twoDigits.format( minute ) + ":" + twoDigits.format( second );
58    }
```

Third overloaded constructor has three `int` arguments

Fourth overloaded constructor has `Time2` argument

Time2.java

Lines 30-33
Third overloaded constructor has three `int` arguments

Lines 36-40
Fourth overloaded constructor has `Time2` argument

Time2.java

```java

   // convert to String in standard-time format
   public String toStandardString()
   {
      DecimalFormat twoDigits = new DecimalFormat( "00" );

      return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) + ":" +
         twoDigits.format( minute ) + ":" + twoDigits.format( second ) +
         ( hour < 12 ? " AM" : " PM" );
   }

} // end class Time2
```

```
1  // Fig. 8.6: TimeTest3.java
2  // Overloaded constructors used to initialize Time2 objects.
3  import javax.swing.*;
4
5  public class TimeTest3 {
6
7     public static void main( String args[] )
8     {
9        Time2 t1 = new Time2();              // 00:00:00
10       Time2 t2 = new Time2( 2 );           // 02:00:00
11       Time2 t3 = new Time2( 21, 34 );      // 21:34:00
12       Time2 t4 = new Time2( 12, 25, 42 );  // 12:25:42
13       Time2 t5 = new Time2( 27, 74, 99 );  // 00:00:00
14       Time2 t6 = new Time2( t4 );          // 12:25:42
15
16       String output = "Constructed with: " +
17          "\nt1: all arguments defaulted" +
18          "\n      " + t1.toUniversalString() +
19          "\n      " + t1.toStandardString();
20
21       output += "\nt2: hour specified; minute and second defaulted" +
22          "\n      " + t2.toUniversalString() +
23          "\n      " + t2.toStandardString();
24
25       output += "\nt3: hour and minute specified; second defaulted" +
26          "\n      " + t3.toUniversalString() +
27          "\n      " + t3.toStandardString();
```

Instantiate each `Time2` reference using a different constructor

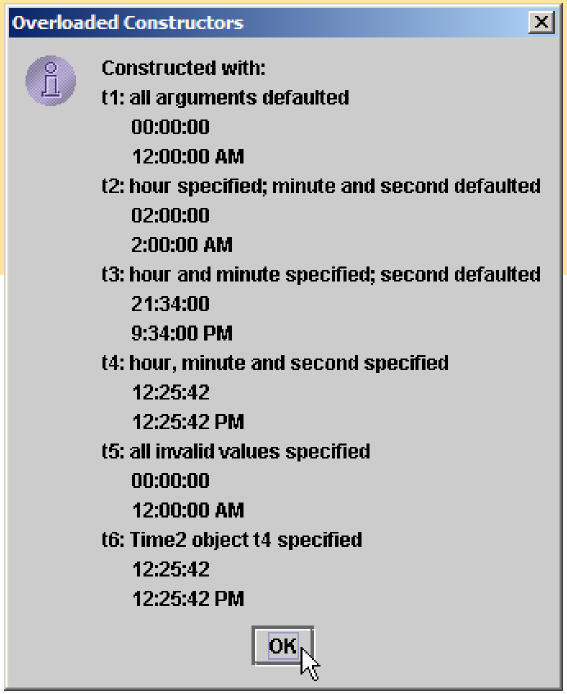
Outline

`TimeTest3.java`

9-14

tiate each `Time2` reference using a different constructor

```

      output += "\nt4: hour, minute and second specified" +
         "\n      " + t4.toUniversalString() +
         "\n      " + t4.toStandardString();

      output += "\nt5: all invalid values specified" +
         "\n      " + t5.toUniversalString() +
         "\n      " + t5.toStandardString();

      output += "\nt6: Time2 object t4 specified" +
         "\n      " + t6.toUniversalString() +
         "\n      " + t6.toStandardString();

      JOptionPane.showMessageDialog( null, output,
         "Overloaded Constructors", JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );

   } // end main

} // end class TimeTest3
```



```
Constructed with:
t1: all arguments defaulted
      00:00:00
      12:00:00 AM
t2: hour specified; minute and second defaulted
      02:00:00
      2:00:00 AM
t3: hour and minute specified; second defaulted
      21:34:00
      9:34:00 PM
t4: hour, minute and second specified
      12:25:42
      12:25:42 PM
t5: all invalid values specified
      00:00:00
      12:00:00 AM
t6: Time2 object t4 specified
      12:25:42
      12:25:42 PM
```

8.8 Using Set and Get Methods

- Accessor method ("get" method)
  - `public` method
  - Allow clients to read `private` data
- Mutator method ("set" method)
  - `public` method
  - Allow clients to modify `private` data

Outline

```
1   // Fig. 8.7: Time3.java
2   // Time3 class declaration with set and get methods.
3   import java.text.DecimalFormat;
4
5   public class Time3 {
6      private int hour;     // 0 - 23
7      private int minute;   // 0 - 59
8      private int second;   // 0 - 59
9
10     // Time3 constructor initializes each
11     // ensures that Time object starts in a consistent state
12     public Time3()
13     {
14        this( 0, 0, 0 ); // invoke Time3 constructor with three arguments
15     }
16
17     // Time3 constructor: hour supplied, minute and second defaulted to 0
18     public Time3( int h )
19     {
20        this( h, 0, 0 ); // invoke Time3 constructor with three arguments
21     }
22
23     // Time3 constructor: hour and minute supplied, second defaulted to 0
24     public Time3( int h, int m )
25     {
26        this( h, m, 0 ); // invoke Time3 constructor with three arguments
27     }
28
```

`private` variables cannot be accessed directly by objects in different classes

`Time3.java`

Lines 6-8

`private` variables cannot be accessed directly by objects in different classes

```
29       // Time3 constructor: hour, minute and second supplied
30       public Time3( int h, int m, int s )
31       {
32          setTime( h, m, s );
33       }
34
35       // Time3 constructor: another Time3 object supplied
36       public Time3( Time3 time )
37       {
38          // invoke Time3 constructor with three arguments
39          this( time.getHour(), time.getMinute(), time.getSecond() );
40       }
41
42       // Set Methods
43       // set a new time value using universal time; perform
44       // validity checks on data; set invalid values to zero
45       public void setTime( int h, int m, int s )
46       {
47          setHour( h );   // set the hour
48          setMinute( m ); // set the minute
49          setSecond( s ); // set the second
50       }
51
52       // validate and set hour
53       public void setHour( int h )
54       {
55          hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
56       }
57
```

Set methods allows objects to manipulate `private` variables

`Time3.java`

Lines 45-68

Set methods allows objects to manipulate `private` variables

```
58       // validate and set minute
59       public void setMinute( int m )
60       {
61          minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
62       }
63
64       // validate and set second
65       public void setSecond( int s )
66       {
67          second = ( ( s >= 0 && s < 60 ) ? s : 0 );
68       }
69
70       // Get Methods
71       // get hour value
72       public int getHour()
73       {
74          return hour;
75       }
76
77       // get minute value
78       public int getMinute()
79       {
80          return minute;
81       }
82
```

Get methods allow objects to read `private` variables

Outline

`Time3.java`

Lines 72-87
Get methods allow objects to read `private` variables

Time3.java

```java
   // get second value
   public int getSecond()
   {
      return second;
   }

   // convert to String in universal-time format
   public String toUniversalString()
   {
      DecimalFormat twoDigits = new DecimalFormat( "00" );

      return twoDigits.format( getHour() ) + ":" +
         twoDigits.format( getMinute() ) + ":" +
         twoDigits.format( getSecond() );
   }

   // convert to String in standard-time format
   public String toStandardString()
   {
      DecimalFormat twoDigits = new DecimalFormat( "00" );

      return ( ( getHour() == 12 || getHour() == 0 ) ?
         12 : getHour() % 12 ) + ":" + twoDigits.format( getMinute() ) +
         ":" + twoDigits.format( getSecond() ) +
         ( getHour() < 12 ? " AM" : " PM" );
   }

} // end class Time3
```

```
1   // Fig. 8.8: TimeTest4.java
2   // Demonstrating the Time3 class set and get methods.
3   import java.awt.*;
4   import java.awt.event.*;
5   import javax.swing.*;
6
7   public class TimeTest4 extends JApplet implements ActionListener {
8      private Time3 time;
9
10     private JLabel hourLabel, minuteLabel,
11     private JTextField hourField, minuteF                            ield;
12     private JButton tickButton;
13
14     // create Time3 object and set up GUI
15     public void init()
16     {
17        time = new Time3();  // create Time3 object
18
19        // get applet's content pane and change its layout to FlowLayout
20        Container container = getContentPane();
21        container.setLayout( new FlowLayout() );
22
23        // set up hourLabel and hourField
24        hourLabel = new JLabel( "Set Hour" );
25        hourField = new JTextField( 10 );
26        container.add( hourLabel );
27        container.add( hourField );
28
```

Declare and instantiate `Time3` object

JTextFields allow user to specify hour

`TimeTest4.java`

Lines 8 and 17
Declare and instantiate `Time3` object

Lines 25 and 31
JTextFields allow user to specify hour.

```
29          // set up minuteLabel and minuteField
30          minuteLabel = new JLabel( "Set Minute" );
31          minuteField = new JTextField( 10 );
32          container.add( minuteLabel );
33          container.add( minuteField );
34
35          // set up secondLabel and secondField
36          secondLabel = new JLabel( "Set Second" );
37          secondField = new JTextField( 10 );
38          container.add( secondLabel );
39          container.add( secondField );
40
41          // set up displayField
42          displayField = new JTextField( 30 );
43          displayField.setEditable( false );
44          container.add( displayField );
45
46          // set up tickButton
47          tickButton = new JButton( "Add 1 to Second" );
48          container.add( tickButton );
49
50          // register event handlers; this applet is the ActionListener,
51          // which contains method actionPerformed that will be called to
52          // handle action events generated by hourField, minuteField,
53          // secondField and tickButton
54          hourField.addActionListener( this );
55          minuteField.addActionListener( this );
56          secondField.addActionListener( this );
57          tickButton.addActionListener( this );
```

`JTextField` allows user to specify minute

`JTextField` allows user to specify second

Outline

`TimeTest4.java`

Line 31
`JTextField` allows user to specify minute

Line 37
`JTextField` allows user to specify second

```
58
59          displayTime(); // update text in displayField and status bar
60
61       } // end method init
62
63       // event handler for button and textfield events
64       public void actionPerformed( ActionEvent event )
65       {
66          // process tickButton event
67          if ( event.getSource() == tickButton )
68             tick();
69
70          // process hourField event
71          else if ( event.getSource() == hourField ) {
72             time.setHour( Integer.parseInt( event.getActionCommand() ) );
73             hourField.setText( "" );
74          }
75
76          // process minuteField event
77          else if ( event.getSource() == minuteField ) {
78             time.setMinute( Integer.parseInt( event.getActionCommand() ) );
79             minuteField.setText( "" );
80          }
81
82          // process secondField event
83          else if ( event.getSource() == secondField ) {
84             time.setSecond( Integer.parseInt( event.getActionCommand() ) );
85             secondField.setText( "" );
86          }
```

TimeTest5 uses Time3 set methods to set Time3 private variables

`TimeTest4.java`

Lines 71-74

Lines 77-80

Lines 83-86

`TimeTest5` uses `Time3` set methods to set `Time3` `private` variables

```
87
88           displayTime(); // update text in displayField and status bar
89
90        } // end method actionPerformed
91
92       // update displayField and applet container's status bar
93       public void displayTime()
94       {
95          displayField.setText( "Hour: " + time.getHour() + "; Minute: " +
96             time.getMinute() + "; Second: " + time.getSecond() );
97
98          showStatus( "Standard time is: " + time.toStandardString() +
99             "; Universal time is: " + time.toUniversalString() );
100
101      } // end method updateDisplay
102
103      // add one to second and update hour/minute if nece
104      public void tick()
105      {
106         time.setSecond( ( time.getSecond() + 1 ) % 60 );
107
108         if ( time.getSecond() == 0 ) {
109            time.setMinute( ( time.getMinute() + 1 ) % 60 );
110
111            if ( time.getMinute() == 0 )
112               time.setHour( ( time.getHour() + 1 ) % 24 );
113         }
114
115      } // end method tick
116
117   } // end class TimeTest4
```

TimeTest5 uses Time3 get methods to read Time3 private variables

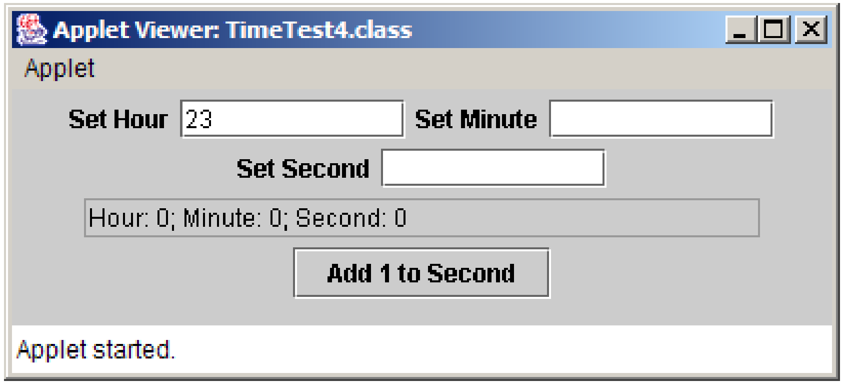
Outline

TimeTest4.java

Lines 95-96

TimeTest5 uses Time3 get methods to read Time3 private variables

TimeTest4.java



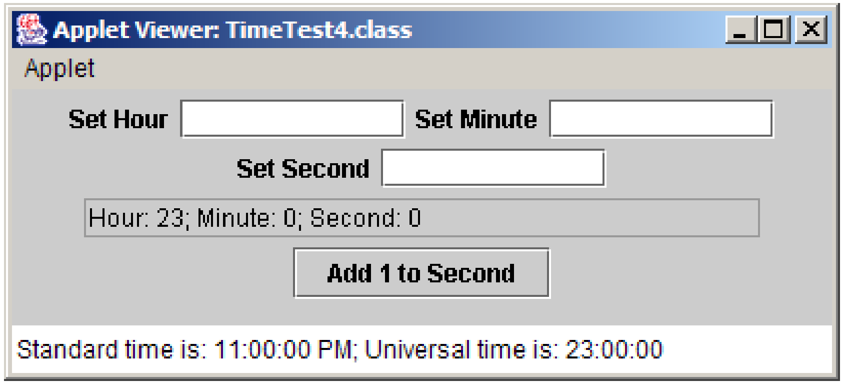
Set Hour 23

Set Minute

Set Second

Hour: 0; Minute: 0; Second: 0

Applet started.



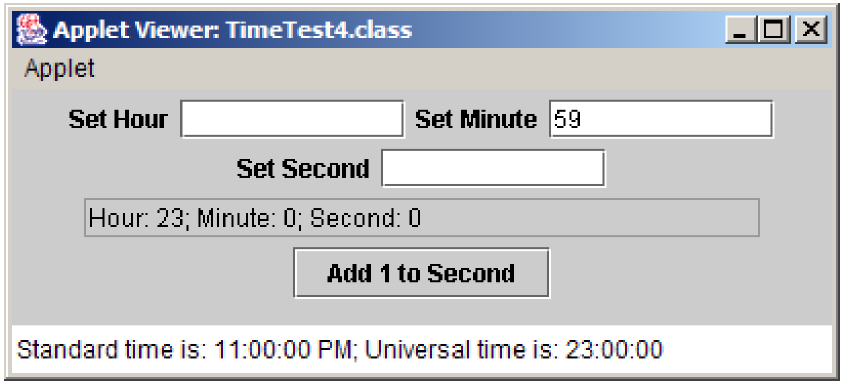
Set Hour

Set Minute

Set Second

Hour: 23; Minute: 0; Second: 0

Standard time is: 11:00:00 PM; Universal time is: 23:00:00



Set Hour

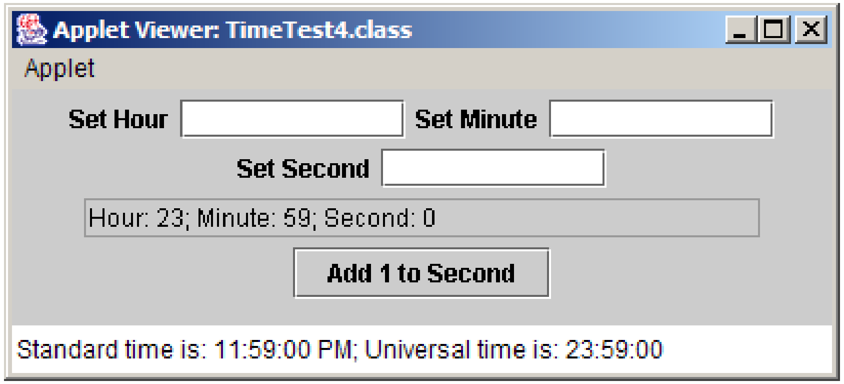
Set Minute 59

Set Second

Hour: 23; Minute: 0; Second: 0

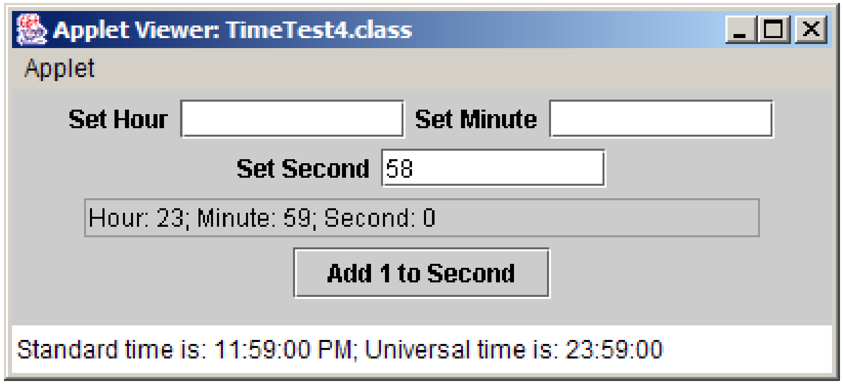
Standard time is: 11:00:00 PM; Universal time is: 23:00:00

TimeTest4.java



Hour: 23; Minute: 59; Second: 0

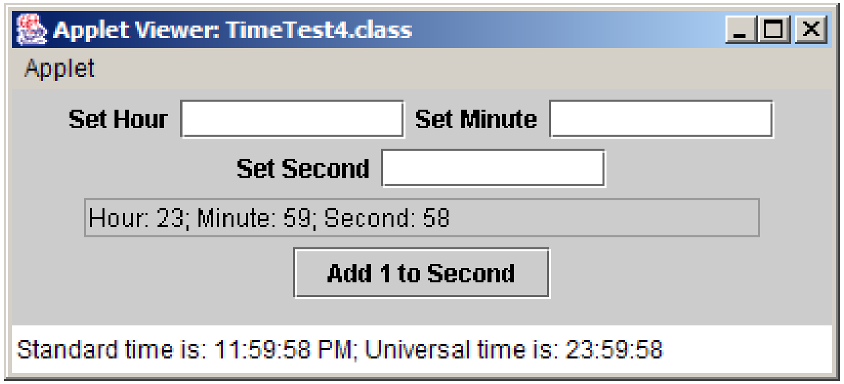
Standard time is: 11:59:00 PM; Universal time is: 23:59:00



Set Second 58

Hour: 23; Minute: 59; Second: 0

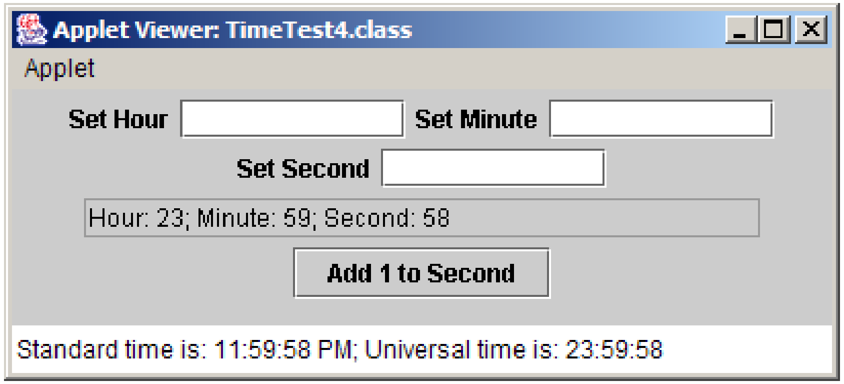
Standard time is: 11:59:00 PM; Universal time is: 23:59:00



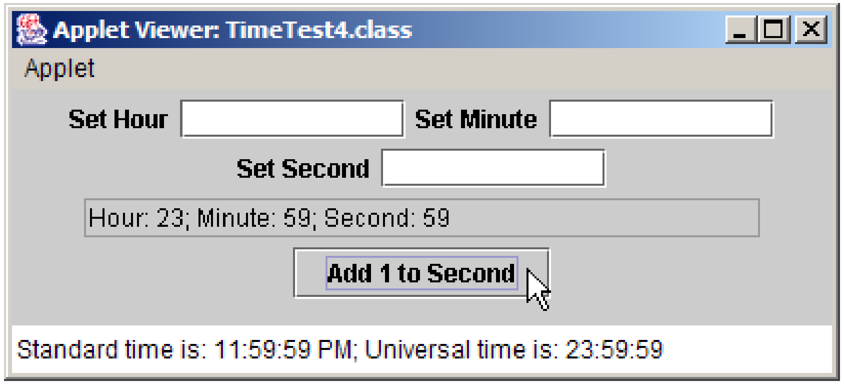
Hour: 23; Minute: 59; Second: 58

Standard time is: 11:59:58 PM; Universal time is: 23:59:58

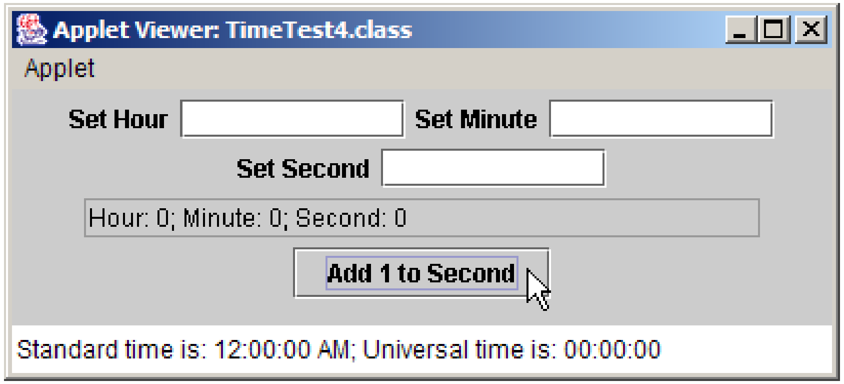
TimeTest4.java



```
Hour: 23; Minute: 59; Second: 58
Standard time is: 11:59:58 PM; Universal time is: 23:59:58
```



```
Hour: 23; Minute: 59; Second: 59
Standard time is: 11:59:59 PM; Universal time is: 23:59:59
```



```
Hour: 0; Minute: 0; Second: 0
Standard time is: 12:00:00 AM; Universal time is: 00:00:00
```

8.9 Composition

- Composition
  - Class contains references to objects of other classes
    - These references are members

```java
1   // Fig. 8.9: Date.java
2   // Date class declaration.
3
4   public class Date {
5      private int month;  // 1-12
6      private int day;    // 1-31 based on month
7      private int year;   // any year
8
9      // constructor: call checkMonth to confirm proper value for month;
10     // call checkDay to confirm proper value for day
11     public Date( int theMonth, int theDay, int theYear )
12     {
13        month = checkMonth( theMonth ); // validate month
14        year = theYear;                 // could validate year
15        day = checkDay( theDay );       // validate d
16
17        System.out.println( "Date object constructor
18           toDateString() );
19
20     } // end Date constructor
21
22     // utility method to confirm proper month value
23     private int checkMonth( int testMonth )
24     {
25        if ( testMonth > 0 && testMonth <= 12 )  // validate month
26           return testMonth;
```

Class `Date` encapsulates data that describes date

`Date` constructor instantiates `Date` object based on specified arguments

Outline

`Date.java`

Line 4
Class `Date` encapsulates data that describes date

Lines 11-20
`Date` constructor instantiates `Date` ased on d arguments

```java

      else { // month is invalid
         System.out.println( "Invalid month (" + testMonth +
            ") set to 1." );
         return 1;  // maintain object in consistent state
      }

   } // end method checkMonth

   // utility method to confirm proper day value based on month and year
   private int checkDay( int testDay )
   {
      int daysPerMonth[] =
         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

      // check if day in range for month
      if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
         return testDay;

      // check for leap year
      if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
           ( year % 4 == 0 && year % 100 != 0 ) ) )
         return testDay;

      System.out.println( "Invalid day (" + testDay + ") set to 1." );

      return 1;  // maintain object in consistent state

   } // end method checkDay
```

Outline

Date.java

```

   // return a String of the form month/day/year
   public String toDateString()
   {
      return month + "/" + day + "/" + year;
   }

} // end class Date
```

Date.java

```
1   // Fig. 8.10: Employee.java
2   // Employee class declaration.
3
4   public class Employee {
5      private String firstName;
6      private String lastName;
7      private Date birthDate;
8      private Date hireDate;
9
10     // constructor to initialize name, birth date and hire date
11     public Employee( String first, String last, Date dateOfBirth,
12        Date dateOfHire )
13     {
14        firstName = first;
15        lastName = last;
16        birthDate = dateOfBirth;
17        hireDate = dateOfHire;
18     }
19
20     // convert Employee to String format
21     public String toEmployeeString()
22     {
23        return lastName + ", " + firstName +
24           "  Hired: " + hireDate.toDateString() +
25           "  Birthday: " + birthDate.toDateString();
26     }
27
28  } // end class Employee
```

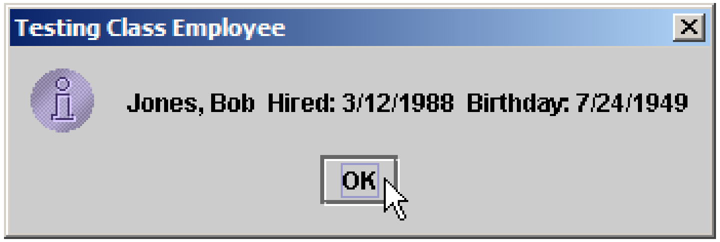
Employee is composed of two references to `Date` objects

`Employee.java`

Lines 7-8

Employee is composed of two references to `Date` objects

```java
// Fig. 8.11: EmployeeTest.java
// Demonstrating an object with a member object.
import javax.swing.JOptionPane;

public class EmployeeTest {

   public static void main( String args[] )
   {
      Date birth = new Date( 7, 24, 1949 );
      Date hire = new Date( 3, 12, 1988 );
      Employee employee = new Employee( "Bob", "Jones", birth, hire );

      JOptionPane.showMessageDialog( null, employee.toEmployeeString(),
         "Testing Class Employee", JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   }

} // end class EmployeeTest
```



Testing Class Employee

Jones, Bob Hired: 3/12/1988 Birthday: 7/24/1949

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
```

Outline

EmployeeTest.java

8.10 Garbage Collection

- Garbage collection
  - Returns memory to system
  - Java performs this automatically
    - object marked for garbage collection if no references to object
- Finalizer method
  - Returns resources to system
  - Java provides method `finalize`
    - Defined in `java.lang.Object`
    - Receives no parameters
    - Returns `void`

8.11 Static Class Members

- `static` keyword
  - `static` class variable
    - Class-wide information
      - All class objects share same data
- Access to a class's `public static` members
  - Qualify the member name with the class name and a dot (.)
    - e.g., Math.random()

```
1   // Fig. 8.12: Employee.java
2   // Employee class declaration.
3   public class Employee {
4      private String firstName;
5      private String lastName;
6      private static int count = 0; // number of objects in memory
7
8      // initialize employee, add 1 to static count and
9      // output String indicating that constructor was called
10     public Employee( String first, String last )
11     {
12        firstName = first;
13        lastName = last;
14
15        ++count;  // increment static count of employees
16        System.out.println( "Employee constructor: " +
17           firstName + " " + lastName );
18     }
19
20     // subtract 1 from static count when garba
21     // calls finalize to clean up object and output String
22     // indicating that finalize was called
23     protected void finalize()
24     {
25        --count;  // decrement static count of employees
26        System.out.println( "Employee finalizer: " +
27           firstName + " " + lastName + "; count = " + count );
28     }
29
```

`Employee` objects share one instance of `count`

Called when `Employee` is marked for garbage collection

Outline

ployee.java

Line 6
`Employee` objects share one instance of `count`

Lines 23-28
Called when `Employee` is marked for garbage collection

```
30    // get first name
31    public String getFirstName()
32    {
33       return firstName;
34    }
35
36    // get last name
37    public String getLastName()
38    {
39       return lastName;
40    }
41
42    // static method to get static count value
43    public static int getCount()
44    {
45       return count;
46    }
47
48 } // end class Employee
```

`static` method accesses `static` variable `count`

`Employee.java`

Lines 43-46
`static` method accesses `static` variable `count`

```
1     // Fig. 8.13: EmployeeTest.java
2     // Test Employee class with static class variable,
3     // static class method, and dynamic memory.
4     import javax.swing.*;
5
6     public class EmployeeTest {
7
8        public static void main( String args[] )
9        {
10          // prove that count is 0 before creating Employees
11          String output = "Employees before instantiation: " +
12             Employee.getCount();
13
14          // create two Employees; count should be 2
15          Employee e1 = new Employee( "Susan", "Baker" );
16          Employee e2 = new Employee( "Bob", "Jones" );
17
18          // prove that count is 2 after creating two Employees
19          output += "\n\nEmployees after instantiation: " +
20             "\nvia e1.getCount(): " + e1.getCount() +
21             "\nvia e2.getCount(): " + e2.getCount() +
22             "\nvia Employee.getCount(): " + Employee.getCount();
23
24          // get names of Employees
25          output += "\n\nEmployee 1: " + e1.getFirstName() +
26             " " + e1.getLastName() + "\nEmployee 2: " +
27             e2.getFirstName() + " " + e2.getLastName();
28
```

EmployeeTest can invoke Employee static method, even though Employee has not been instantiated

EmployeeTest.java

Line 12
EmployeeTest can invoke Employee static method, even though Employee has not been instantiated

Outline

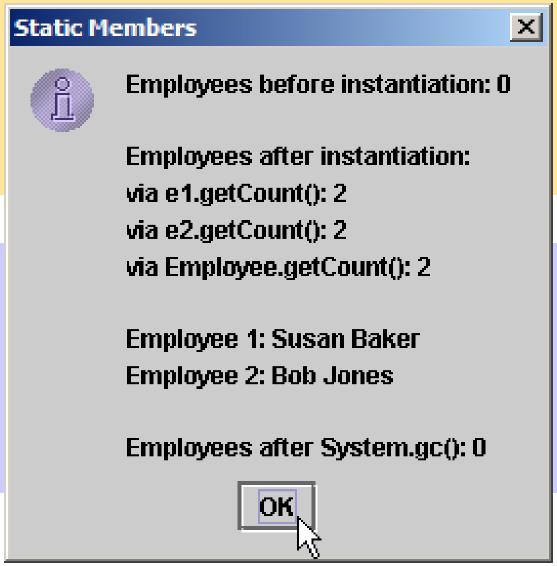
EmployeeTest.java

```
29          // decrement reference count for each Employee object; in this
30          // example, there is only one reference to each Employee, so these
31          // statements mark each                                     tion
32          e1 = null;
33          e2 = null;
34
35          System.gc(); // suggest call to garbage collector
36
37          // show Employee count after calling garbage collector; count
38          // displayed may be 0, 1 or 2 based on whether garbage collector
39          // executes immediately and number of Employee objects collected
40          output += "\n\nEmployees after System.gc(): " +
41             Employee.getCount();
42
43          JOptionPane.showMessageDialog( null, output,
44             "Static Members", JOptionPane.INFORMATION_MESSAGE );
45
46          System.exit( 0 );
47       }
48
49    } // end class EmployeeTest
```

Calls Java’s automatic garbage-collection mechanism

Line 35
Calls Java’s automatic garbage-collection mechanism

```
Employee constructor: Susan Baker
Employee constructor: Bob Jones
Employee finalizer: Susan Baker; count = 1
Employee finalizer: Bob Jones; count = 0
```



Employees before instantiation: 0

Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Jones

Employees after System.gc(): 0

8.12 Final Instance Variables

- `final` keyword
  - Indicates that variable is not modifiable
    - Any attempt to modify `final` variable results in error

      ```
      private final int INCREMENT = 5;
      ```

    - Declares variable `INCREMENT` as a *constant*
  - Enforces *principle of least privilege*

```java
// Fig. 8.14: IncrementTest.java
// Initializing a final variable.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class IncrementTest extends JApplet implements ActionListener {
   private Increment incrementObject;
   private JButton button;

   // set up GUI
   public void init()
   {
      incrementObject = new Increment( 5 );

      Container container = getContentPane();

      button = new JButton( "Click to increment" );
      button.addActionListener( this );
      container.add( button );
   }

   // add INCREMENT to total when user clicks button
   public void actionPerformed( ActionEvent actionEvent )
   {
      incrementObject.increment();
      showStatus( incrementObject.toIncrementString() );
   }

} // end class IncrementTest

```

IncrementTest.java

Outline

```
32    // class containing constant variable
33    class Increment {
34       private int count = 0;        // number of increments
35       private int total = 0;        // total of all inc
36       private final int INCREMENT; // constant variabl
37
38       // initialize constant INCREMENT
39       public Increment( int incrementValue )
40       {
41          INCREMENT = incrementValue; // intialize constant variable (once)
42       }
43
44       // add INCREMENT to total and add 1 to count
45       public void increment()
46       {
47          total += INCREMENT;
48          ++count;
49       }
50
51       // return String representation of an Increment object's data
52       public String toIncrementString()
53       {
54          return "After increment " + count + ": total = " + total;
55       }
56
57    } // end class Increment
```

`final` keyword declares `INCREMENT` as constant

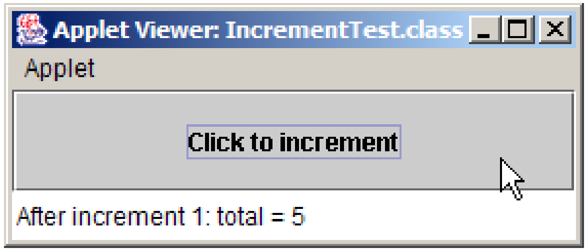
`final` variable `INCREMENT` must be initialized before using it

crement.java

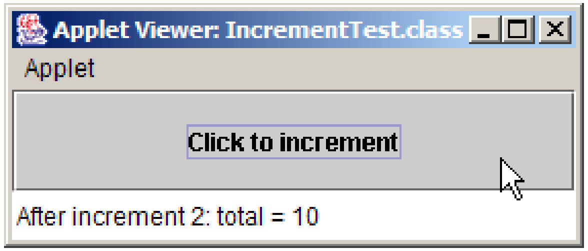
Line 36
`final` keyword declares `INCREMENT` as constant

Line 41
`final` variable `INCREMENT` must be initialized before using it

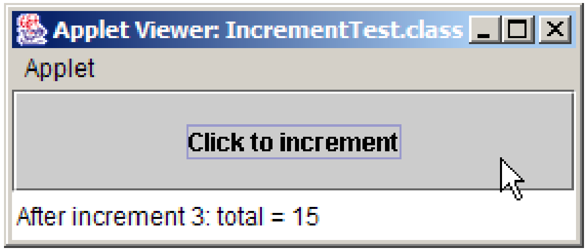
```
IncrementTest.java:40: variable INCREMENT might not have been initialized
   {
   ^
1 error
```



After increment 1: total = 5



After increment 2: total = 10



After increment 3: total = 15

8.13 Creating Packages

- We can `import` *packages* into programs
  - Group of related classes and interfaces
  - Help manage complexity of application components
  - Facilitate software reuse
  - Provide convention for unique class names
    - Popular package-naming convention
      - Reverse Internet domain name
        - e.g., `com.deitel`

Outline

```
1  // Fig. 8.16: Time1.java
2  // Time1 class declaration maintains the time
3  package com.deitel.jhtp5.ch08;
4
5  import java.text.DecimalFormat;
6
7  public class Time1 extends Object {
8     private int hour;     // 0 - 23
9     private int minute;   // 0 - 59
10   private int second;   // 0 - 59
11
12   // Time1 constructor initializes each instance variable to zero;
13   // ensures that each Tim                                      te
14   public Time1()
15   {
16      setTime( 0, 0, 0 );
17   }
18
19   // set a new time value using universal time; perform
20   // validity checks on the data; set invalid values to zero
21   public void setTime( int h, int m, int s )
22   {
23      hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
24      minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
25      second = ( ( s >= 0 && s < 60 ) ? s : 0 );
26   }
27
```

Class `Time1` is placed in this `package`

Class `Time1` is in directory `com/deitel/jhtp5/ch08`

`import` class `DecimalFormat` from `package java.text`

`Time1.java`

Line 3
Class `Time1` is placed in this `package`

Line 3
Class `Time1` is in directory `com/deitel/jhtp5/ch08`

Line 5
`import` class `DecimalFormat` from `package java.text`

```java
28    // convert to String in universal-time format
29    public String toUniversalString()
30    {
31       DecimalFormat twoDigits = new DecimalFormat( "00" );
32
33       return twoDigits.format( hour ) + ":" +
34          twoDigits.format( minute ) + ":" + twoDigits.format( second );
35    }
36
37    // convert to String in standard-time format
38    public String toStandardString()
39    {
40       DecimalFormat twoDigits = new DecimalFormat( "00" );
41
42       return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) + ":" +
43          twoDigits.format( minute ) + ":" + twoDigits.format( second ) +
44          ( hour < 12 ? " AM" : " PM" );
45    }
46
47 } // end class Time1
```

`DecimalFormat` from package `java.text`

Line 31
`DecimalFormat` from package `java.text`

```
1     // Fig. 8.17: TimeTest1.java
2     // Class TimeTest1 to exercise class Time1.
3
4     // Java packages
5     import javax.swing.JOptionPane;
6
7     // Deitel packages
8     import com.deitel.jhtp5.ch08.Time1;  // import Time1 class
9
10    public class TimeTest1 {
11
12       public static void main( String arg
13       {
14          Time1 time = new Time1();  // calls Time1 constructor
15
16          // append String version of time to String output
17          String output = "The initial universal time is: " +
18             time.toUniversalString() + "\nThe initial standard time is: " +
19             time.toStandardString();
20
21          // change time and append updated time to output
22          time.setTime( 13, 27, 6 );
23          output += "\n\nUniversal time after setTime is: " +
24             time.toUniversalString() +
25             "\nStandard time after setTime is: " + time.toStandardString();
26
```

import class JOptionPane from package javax.swing

import class Time1 from package com.deitel.jhtp4.ch08

TimeTest1 can declare Time1 object

TimeTest1.java

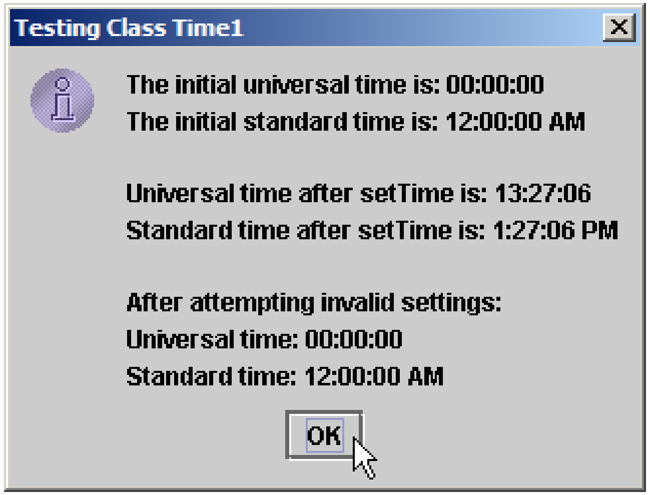
Line 5
import class JOptionPane from package javax.swing

Line 8
import class Time1 from package com.deitel.jhtp4.ch08

Line 14
TimeTest1 can declare Time1 object

```
      // set time with invalid values; append updated time to output
      time.setTime( 99, 99, 99 );
      output += "\n\nAfter attempting invalid settings: " +
         "\nUniversal time: " + time.toUniversalString() +
         "\nStandard time: " + time.toStandardString();

      JOptionPane.showMessageDialog( null, output,
         "Testing Class Time1", JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );

   } // end main

} // end class TimeTest1
```

TimeTest1.java



```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

8.14 Package Access

- Package access
  - Variable or method does not have member access modifier

Outline

```
1   // Fig. 8.18: PackageDataTest.java
2   // Classes in the same package (i.e., the same directory) can
3   // use package access data of other classes in the sa
4   import javax.swing.JOptionPane;
5
6   public class PackageDataTest {
7
8      public static void main( String args[] )
9      {
10        PackageData packageData = new PackageData();
11
12        // append String representation of packageData to output
13        String output = "After instantiation:\n" +
14           packageData.toPackageDataString();
15
16        // change package access data in packageData ob
17        packageData.number = 77;
18        packageData.string = "Goodbye";
19
20        // append String representation of packageData to output
21        output += "\nAfter changing values:\n" +
22           packageData.toPackageDataString();
23
24        JOptionPane.showMessageDialog( null, output, "Package Access",
25           JOptionPane.INFORMATION_MESSAGE );
26
```

Instantiate reference to `PackageData` object

`PackageDataTest` can access `PackageData` data, because each class shares same `package`

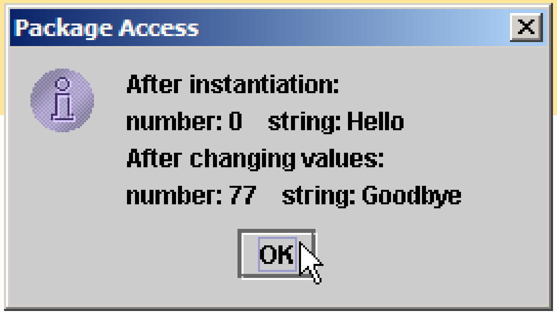
kageDataTest
va

Line 10
Instantiate reference to `PackageData` object

Lines 13-22
geDataTest
ess
geData data,
e each class
shares same `package`

```
27          System.exit( 0 );
28       }
29
30    } // end class PackageDataTest
31
32    // class with package access instance variables
33    class PackageData {
34       int number;      // package-access instance variable
35       String string;   // package-access instance variable
36
37       // constructor
38       public PackageData()
39       {
40          number = 0;
41          string = "Hello";
42       }
43
44       // return PackageData object String representation
45       public String toPackageDataString()
46       {
47          return "number: " + number + "    string: " + string;
48       }
49
50    } // end class PackageData
```

No access modifier, so class has package-access variables


Package Access

After instantiation:
number: 0   string: Hello
After changing values:
number: 77   string: Goodbye

Outline

PackageDataTest
.java

Line 33
No access modifier, so class has package-
ss variables

8.15 Software Reusability

- Java
  - Framework for achieving software reusability
  - Rapid applications development (RAD)
    - e.g., creating a GUI application quickly

8.16 Data Abstraction and Encapsulation

- Information hiding
  - Stack data structure
    - Last in-first out (LIFO)
    - Developer creates stack
      - Hides stack’s implementation details from clients
    - Data abstraction
      - Abstract data types (ADTs)

8.16 Data Abstraction and Encapsulation (Cont.)

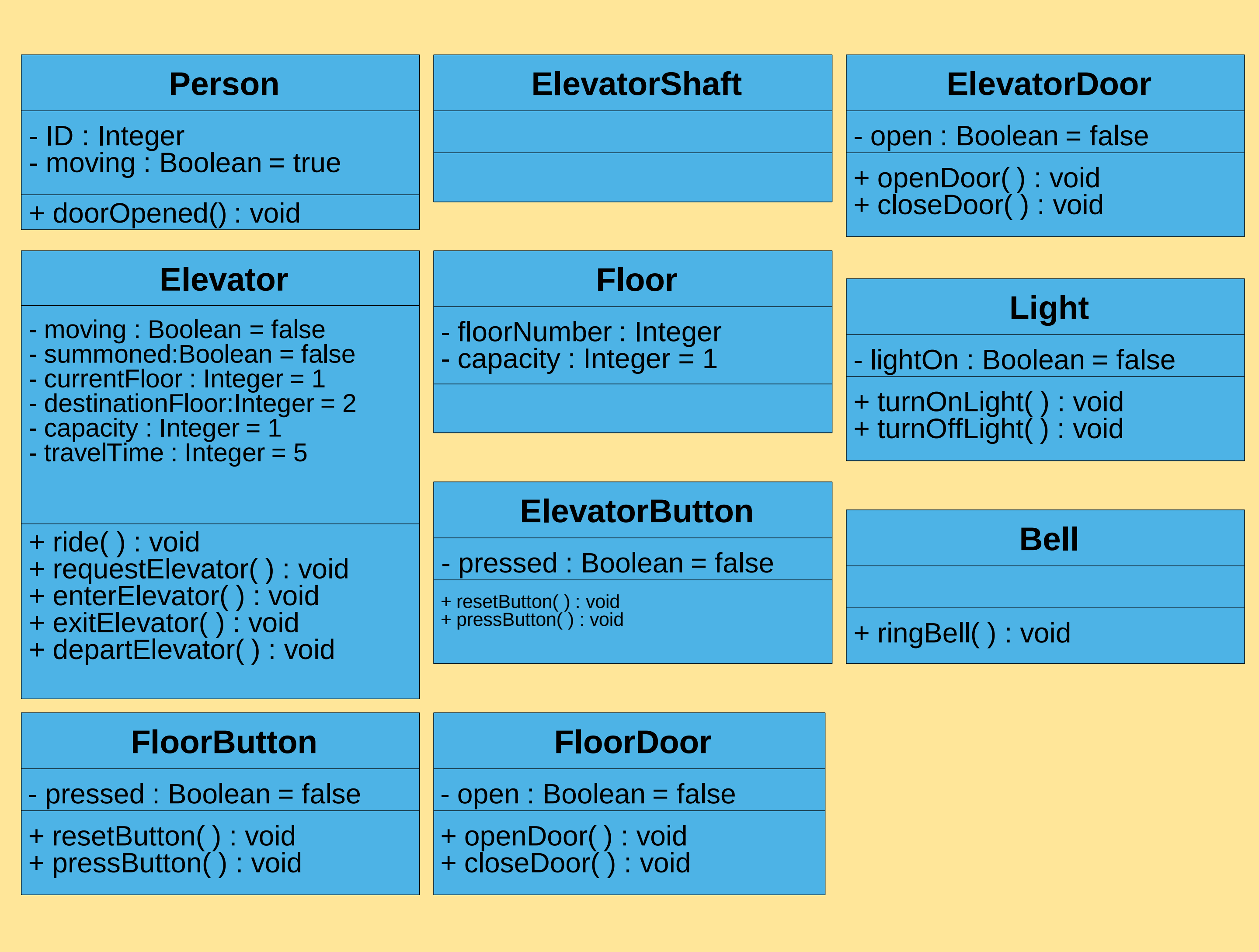
- Abstract Data Type (ADT)
  - Queue
    - Line at grocery store
    - First-in, first-out (FIFO)
      - Enqueue to place objects in queue
      - Dequeue to remove object from queue
      - Enqueue and dequeue hide internal data representation

8.17 (Optional Case Study) Thinking About Objects: Starting to Program the Classes for the Elevator Simulation

- Visibility
  - Apply member-access modifiers to class members
  - `public` methods
    - to provide services to clients
  - `private` variables
    - To promote encapsulation

8.17 Thinking About Objects (cont.)

- Class diagram (UML)
  - Member-access modifiers
    - `public`
      - Denoted by plus sign (`+`)
    - `private`
      - Denoted by minus sign (`-`)



Person

- ID : Integer
- moving : Boolean = true

+ doorOpened() : void

ElevatorShaft

ElevatorDoor

- open : Boolean = false

+ openDoor( ) : void
+ closeDoor( ) : void

Elevator

- moving : Boolean = false
- summoned:Boolean = false
- currentFloor : Integer = 1
- destinationFloor:Integer = 2
- capacity : Integer = 1
- travelTime : Integer = 5

+ ride( ) : void
+ requestElevator( ) : void
+ enterElevator( ) : void
+ exitElevator( ) : void
+ departElevator( ) : void

Floor

- floorNumber : Integer
- capacity : Integer = 1

Light

- lightOn : Boolean = false

+ turnOnLight( ) : void
+ turnOffLight( ) : void

ElevatorButton

- pressed : Boolean = false

+ resetButton( ) : void
+ pressButton( ) : void

Bell

+ ringBell( ) : void

FloorButton

- pressed : Boolean = false

+ resetButton( ) : void
+ pressButton( ) : void

FloorDoor

- open : Boolean = false

+ openDoor( ) : void
+ closeDoor( ) : void

Fig 8.19 Class diagram with visibility notations.

8.17 Thinking About Objects (cont.)

- Navigability
  - Indicate in which direction an association can be navigated
  - Help programmers determine which objects need references to other objects

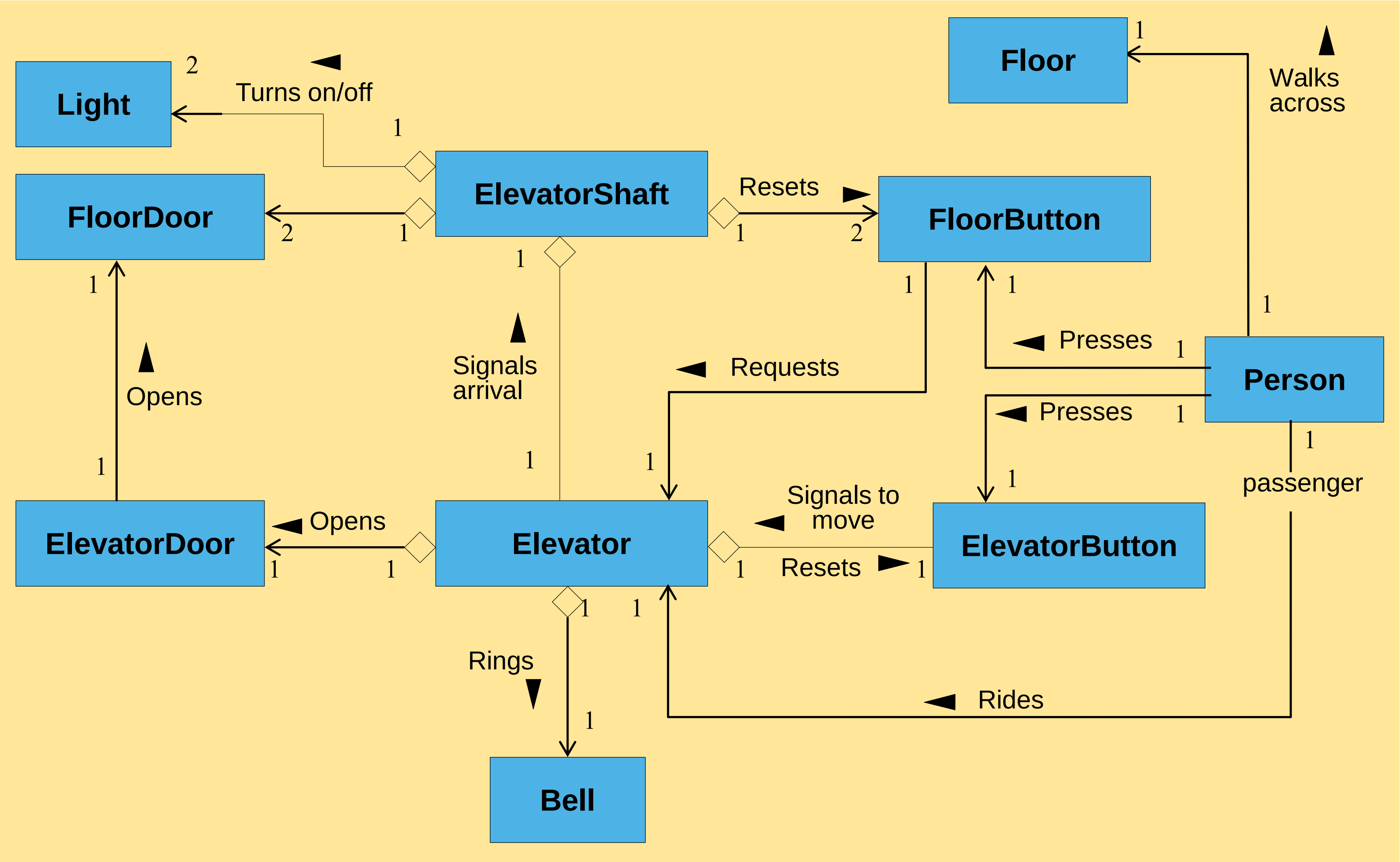


Fig 8.20 Class diagram with navigability.

8.17 Thinking About Objects (cont.)

- Implementation
  - Forward engineering
    - Transform design (i.e., class diagram) to code

8.17 Thinking About Objects (cont.)

- We generate “skeleton code” with our design
  - Use class `Elevator` as example
  - Four steps:
    - Use name in first compartment to declare `public` class
      - Empty constructor
    - Use attributes in second compartment to declare instance variables
    - Use associations in class diagram (Fig. 3.19) to declare object references
    - Use operations in third compartment to declare methods

8.17 Thinking About Objects (cont.) Step 1

```
public class Elevator {

   public Elevator() {}
}
```

8.17 Thinking About Objects (cont.) Step 2

```
public class Elevator {

   // attributes
   private boolean moving;
   private boolean summoned;
   private int currentFloor = 1;
   private int destinationFloor = 2;
   private int capacity = 1;
   private int travelTime = 5;

   // constructor
   public Elevator() {}
}
```

8.17 Thinking About Objects (cont.) Step 3

```
public class Elevator {

   // attributes
   private boolean moving;
   private boolean summoned;
   private int currentFloor = 1;
   private int destinationFloor = 2;
   private int capacity = 1;
   private int travelTime = 5;

   // associated objects
   private ElevatorDoor elevatorDoor;
   private ElevatorButton elevatorButton;
   private Bell bell;

   // constructor
   public Elevator() {}
}
```

8.17 Thinking About Objects (cont.) Step 4

```
public class Elevator {

   // attributes
   private boolean moving;
   private boolean summoned;
   private int currentFloor = 1;
   private int destinationFloor = 2;
   private int capacity = 1;
   private int travelTime = 5;

   // associated objects
   private ElevatorDoor elevatorDoor;
   private ElevatorButton elevatorButton;
   private Bell bell;

   // constructor
   public Elevator() {}

   // operations
   public void ride() {}
   public void requestElevator() {}
   public void enterElevator() {}
   public void exitElevator() {}
   public void departElevator() {}
}
```