8 - 9

Classes and Objects: A Deeper Look into OOP

Objectives

- data abstraction and data hiding (encapsulation)
- inheritance
- composition
- introduction to exceptions

Outline

Outline

Outline

Data Abstraction and Encapsulation

- **Data abstraction**
  - **Abstract data types (ADTs)**
- **Information hiding**
  - **Classes normally hide details of their implementation, only expose public interface to their clients**
  - **Keep variables (properties) private**
  - **Expose public methods to manipulate properties**

How to reuse classes (and code)

- Have to do better than just cut & paste
- Reusing classes
  - Create new class from existing class(es)
  - Absorb existing class data and behaviors
  - Enhance with new capabilities

- Composition (compose using existing classes)
- Inheritance (inherit from existing classes)

Composition

- **Composition**
  - **A class can have references to objects of other classes as members**
  - **Sometimes referred to as a *has-a* relationship**

Inheritance

- **Inheritance**
  - **Subclass extends superclass**
    - **Subclass**
      - **More specialized group of objects**
      - **Behaviors inherited from superclass (can customize)**
      - **Additional behaviors**
  - **Sometimes referred to as a *is-a* relationship**

Case Study: Time and Clock Class

- `public` services (or `public` interface)
  - `public` methods available for a client to use

- If a class does not define a constructor the compiler will provide a default constructor
- Instance variables
  - Can be initialized when they are declared or in a constructor
  - Should maintain consistent (valid) values

Outline

Time1.java

(1 of 2)

```
1  // Fig. 8.1: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3
4  public class Time1
5  {
6     private int hour;   // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // set a new time value using universal time; ensure that
11    // the data remains consistent by setting invalid values to zero
12    public void setTime( int h, int m, int s )
13
14       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
15       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
16       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
17    } // end method setTime
18
```

private instance variables

Declare **public** method **setTime**

Validate parameter values before setting instance variables

Outline

Time1.java

(2 of 2)

```
19    // convert to String in universal-time format (HH:MM:SS)
20    public String toUniversalString()
21    {
22       return String.format( "%02d:%02d:%02d", hour, minute, second );
23    } // end method toUniversalString
24
25    // convert to String in standard-time format (H:MM:SS AM or PM)
26    public String toString()
27    {
28       return String.format( "%d:%02d:%02d %s",
29          ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
30          minute, second, ( hour < 12 ? "AM" : "PM" ) );
31    } // end method toString
32 } // end class Time1
```

format strings

Outline

Time1Test.java

(1 of 2)

```
1  // Fig. 8.2: Time1Test.java
2  // Time1 object used in an application.
3
4  public class Time1Test
5  {
6     public static void main( String args[] )
7     {
8        // create and initialize a Time1 object
9        Time1 time = new Time1(); // invokes Time1 constructor
10
11       // output string representations of the time
12       System.out.print( "The initial universal time is: " );
13       System.out.println( time.toUniversalString() );
14       System.out.print( "The initial standard time is: " );
15       System.out.println( time.toString() );
16       System.out.println(); // output a blank line
17
```

Create a **Time1** object

Call **toUniversalString** method

Call **toString** method

Outline

Time1Test.java

```
18       // change time and output updated time
19       time.setTime( 13, 27, 6 );
20       System.out.print( "Universal time after setTime is: " );
21       System.out.println( time.toUniversalString() );
22       System.out.print( "Standard time after setTime is: " );
23       System.out.println( time.toString() );
24       System.out.println(); // output a blank line
25
26       // set time with invalid values; output updated time
27       time.setTime( 99, 99, 99 );
28       System.out.println( "After attempting invalid settings:" );
29       System.out.print( "Universal time: " );
30       System.out.println( time.toUniversalString() );
31       System.out.print( "Standard time: " );
32       System.out.println( time.toString() );
33    } // end main
34 } // end class Time1Test
```

Call **setTime** method

Call **setTime** method with invalid values

```
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
```

Controlling Access to Members

- A class's public interface
  - `public` methods a view of the services the class provides to the class's clients
- A class's implementation details
  - `private` variables and `private` methods are not accessible to the class's clients

Outline

MemberAccessTest.java

```
1  // Fig. 8.3: MemberAccessTest.java
2  // Private members of class Time1 are not accessible.
3  public class MemberAccessTest
4  {
5     public static void main( String args[] )
6     {
7        Time1 time = new Time1(); // create and initialize Time1 object
8
9        time.hour = 7;     // error: hour has private access in Time1
10       time.minute = 15; // error: minute has private access in Time1
11       time.second = 30; // error: second has private access in Time1
12    } // end main
13 } // end class MemberAccessTest
```

Attempting to access **`private`** instance variables

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

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

- The `this` reference
  - Object can access a reference to itself with keyword `this`
  - Non-`static` methods implicitly use `this` when referring to the object's instance variables and other methods
  - Can be used to access instance variables when they are shadowed by local variables or method parameters

Outline

ThisTest.java

(1 of 2)

```
1  // Fig. 8.4: ThisTest.java
2  // this used implicitly and explicitly to refer to members of an object.
3
4  public class ThisTest
5  {
6     public static void main( String args[] )
7     {
8        SimpleTime time = new SimpleTime( 15, 30, 19 );
9        System.out.println( time.buildString() );
10    } // end main
11 } // end class ThisTest
12
13 // class SimpleTime demonstrates the "this" reference
14 class SimpleTime
15 {
16    private int hour;   // 0-23
17    private int minute; // 0-59
18    private int second; // 0-59
19
20    // if the constructor uses parameter names identical to
21    // instance variable names the "this" reference is
22    // required to distinguish between names
23    public SimpleTime( int hour, int minute, int second )
24    {
25       this.hour = hour;       // set "this" object's hour
26       this.minute = minute;   // set "this" object's minute
27       this.second = second;   // set "this" object's second
28    } // end SimpleTime constructor
29
```

Create new **SimpleTime** object

Declare instance variables

Method parameters shadow instance variables

Using `this` to access the object's instance variables

Outline

ThisTest.java

```
30    // use explicit and implicit "this" to call toUniversalString
31    public String buildString()
32    {
33       return String.format( "%24s: %s\n%24s: %s",
34          "this.toUniversalString()", this.toUniversalString(),
35          "toUniversalString()", toUniversalString() );
36    } // end method buildString
37
38    // convert to String in universal-time format (HH:MM:SS)
39    public String toUniversalString()
40    {
41       // "this" is not required here to access instance variables,
42       // because method does not have local variables with same
43       // names as instance variables
44       return String.format( "%02d:%02d:%02d",
45          this.hour, this.minute, this.second );
46    } // end method toUniversalString
47 } // end class SimpleTime
```

Using **`this`** explicitly and implicitly to call **`toUniversalString`**

Use of **`this`** not necessary here

```
this.toUniversalString(): 15:30:19
     toUniversalString(): 15:30:19
```

(2 of 2)

Example: BinaryTime