Programming Languages: Java

Lecture 4

OOP: Classes and Objects

OOP: Inheritance

OOP: polymorphism

Instructor: Omer Boyaci

Classes and Objects

WE WILL COVER

- Encapsulation and data hiding.
- The notions of data abstraction and abstract data types (ADTs).
- To use keyword `this`.
- To use `static` variables and methods.
- To import `static` members of a class.
- To use the `enum` type to create sets of constants with unique identifiers.
- How to declare `enum` constants with parameters.

- Controlling Access to Members
- Referring to the Current Object's Members with the `this`
- `Time` Class Case Study: Overloaded Constructors
- Default and No-Argument Constructors
- Notes on *Set* and *Get* Methods
- Composition
- Enumerations
- Garbage Collection and Method `finalize`
- `static` Class Members
- `static` Import
- `final` Instance Variables
- Software Reusability
- Data Abstraction and Encapsulation
- `Time` Class Case Study: Creating Packages
- Package Access

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

Software Engineering Observation

Classes simplify programming, because the client can use only the `public` methods exposed by the class. Such methods are usually client oriented rather than implementation oriented. Clients are neither aware of, nor involved in, a class's implementation. Clients generally care about *what* the class does but not *how* the class does it.

Software Engineering Observation 8.3

Interfaces change less frequently than implementations. When an implementation changes, implementation-dependent code must change accordingly. Hiding the implementation reduces the possibility that other program parts will become dependent on class-implementation details.

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

- The `this` reference
  - Any 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
- A `.java` file can contain more than one class
  - But only one class in each `.java` file can be `public`

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

(2 of 2)

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

Time Class Case Study: Overloaded Constructors

- **Overloaded constructors**
  - **Provide multiple constructor definitions with different signatures**
- **No-argument constructor**
  - **A constructor invoked without arguments**
- **The `this` reference can be used to invoke another constructor**
  - **Allowed only as the first statement in a constructor's body**

Outline

Time2.java

(1 of 4)

```
1  // Fig. 8.5: Time2.java
2  // Time2 class declaration with overloaded constructors.
3
4  public class Time2
5  {
6     private int hour;   // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // Time2 no-argument constructor: initializes each instance variable
11    // to zero; ensures that Time2 objects start in a consistent state
12    public Time2()
13    {
14       this( 0, 0, 0 ); // invoke Time2 constructor with three arguments
15    } // end Time2 no-argument constructor
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 constructor with three arguments
21    } // end Time2 one-argument constructor
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    } // end Time2 two-argument constructor
28
```

No-argument constructor

Invoke three-argument constructor

Outline

```
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    } // end Time2 three-argument constructor
34
35    // Time2 constructor: another Time2 object supplied
36    public Time2( Time2 time )
37    {
38       // invoke Time2 three-argument c
39       this( time.getHour(), time.getMinute(), time.getSecond() );
40    } // end Time2 constructor with a Time2 object argument
41
42    // Set Methods
43    // set a new time value using universal time; ensure that
44    // the data remains consistent by setting 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    } // end method setTime
51
```

Call **setTime** method

Time2.java

Constructor takes a reference to another **Time2** object as a parameter

(2 of 4)

Could have directly accessed instance variables of object **time** here

Outline

Time2.java

(3 of 4)

```
52    // validate and set hour
53    public void setHour( int h )
54    {
55       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
56    } // end method setHour
57
58    // validate and set minute
59    public void setMinute( int m )
60    {
61       minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
62    } // end method setMinute
63
64    // validate and set second
65    public void setSecond( int s )
66    {
67       second = ( ( s >= 0 && s < 60 ) ? s : 0 );
68    } // end method setSecond
69
70    // Get Methods
71    // get hour value
72    public int getHour()
73    {
74       return hour;
75    } // end method getHour
76
```

Outline

Time2.java

(4 of 4)

```
77    // get minute value
78    public int getMinute()
79    {
80       return minute;
81    } // end method getMinute
82
83    // get second value
84    public int getSecond()
85    {
86       return second;
87    } // end method getSecond
88
89    // convert to String in universal-time format (HH:MM:SS)
90    public String toUniversalString()
91    {
92       return String.format(
93          "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
94    } // end method toUniversalString
95
96    // convert to String in standard-time format (H:MM:SS AM or PM)
97    public String toString()
98    {
99       return String.format( "%d:%02d:%02d %s",
100         ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
101         getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
102   } // end method toString
103} // end class Time2
```

Software Engineering Observation 8.4

When one object of a class has a reference to another object of the same class, the first object can access all the second object's data and methods (including those that are `private`).

Time Class Case Study: Overloaded Constructors (Cont.)

- **Using *set* methods**
  - **Having constructors use *set* methods to modify instance variables instead of modifying them directly simplifies implementation changing**

Outline

Call overloaded constructors

Time2Test.java

(1 of 3)

```
1  // Fig. 8.6: Time2Test.java
2  // Overloaded constructors used to initialize Time2 objects.
3
4  public class Time2Test
5  {
6     public static void main( String args[] )
7     {
8        Time2 t1 = new Time2();           // 00:00:00
9        Time2 t2 = new Time2( 2 );        // 02:00:00
10       Time2 t3 = new Time2( 21, 34 );   // 21:34:00
11       Time2 t4 = new Time2( 12, 25, 42 ); // 12:25:42
12       Time2 t5 = new Time2( 27, 74, 99 ); // 00:00:00
13       Time2 t6 = new Time2( t4 );       // 12:25:42
14
15       System.out.println( "Constructed with:" );
16       System.out.println( "t1: all arguments defaulted" );
17       System.out.printf( "   %s\n", t1.toUniversalString() );
18       System.out.printf( "   %s\n", t1.toString() );
19
```

```
20      System.out.println(
21         "t2: hour specified; minute and second defaulted" );
22      System.out.printf( "   %s\n", t2.toUniversalString() );
23      System.out.printf( "   %s\n", t2.toString() );
24
25      System.out.println(
26         "t3: hour and minute specified; second defaulted" );
27      System.out.printf( "   %s\n", t3.toUniversalString() );
28      System.out.printf( "   %s\n", t3.toString() );
29
30      System.out.println( "t4: hour, minute and second specified" );
31      System.out.printf( "   %s\n", t4.toUniversalString() );
32      System.out.printf( "   %s\n", t4.toString() );
33
34      System.out.println( "t5: all invalid values specified" );
35      System.out.printf( "   %s\n", t5.toUniversalString() );
36      System.out.printf( "   %s\n", t5.toString() );
37
```

Outline

Time2Test.java

(2 of 3)

```
38       System.out.println( "t6: Time2 object t4 specified" );
39       System.out.printf( "   %s\n", t6.toUniversalString() );
40       System.out.printf( "   %s\n", t6.toString() );
41    } // end main
42 } // end class Time2Test
```

```
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
```

Outline

Time2Test.java

(3 of 3)

Default and No-Argument Constructors

- **Every class must have at least one constructor**
  - **If no constructors are declared, the compiler will create a default constructor**
    - **Takes no arguments and initializes instance variables to their initial values specified in their declaration or to their default values**
      - **Default values are zero for primitive numeric types, `false` for `boolean` values and `null` for references**
  - **If constructors are declared, the default initialization for objects of the class will be performed by a no-argument constructor (if one is declared)**

Notes on *Set* and *Get* Methods

- ***Set*** **methods**
  - **Also known as mutator methods**
  - **Assign values to instance variables**
  - **Should validate new values for instance variables**
    - **Can return a value to indicate invalid data**
- ***Get*** **methods**
  - **Also known as accessor methods or query methods**
  - **Obtain the values of instance variables**
  - **Can control the format of the data it returns**

Notes on *Set* and *Get* Methods (Cont.)

- **Predicate methods**
  - **Test whether a certain condition on the object is true or false and returns the result**
  - **Example: an `isEmpty` method for a container class (a class capable of holding many objects)**
- **Encapsulating specific tasks into their own methods simplifies debugging efforts**

Composition

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

Outline

Date.java

(1 of 3)

```
1  // Fig. 8.7: Date.java
2  // Date class declaration.
3
4  public class Date
5  {
6     private int month; // 1-12
7     private int day;   // 1-31 based on month
8     private int year;  // any year
9
10    // constructor: call checkMonth to confirm proper value for month;
11    // call checkDay to confirm proper value for day
12    public Date( int theMonth, int theDay, int theYear )
13    {
14       month = checkMonth( theMonth ); // validate month
15       year = theYear; // could validate year
16       day = checkDay( theDay ); // validate day
17
18       System.out.printf(
19          "Date object constructor for date %s\n", this );
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;
27       else // month is invalid
28       {
29          System.out.printf(
30             "Invalid month (%d) set to 1.", testMonth );
31          return 1; // maintain object in consistent state
32       } // end else
33    } // end method checkMonth
34
35    // utility method to confirm proper day value based on month and year
36    private int checkDay( int testDay )
37    {
38       int daysPerMonth[] =
39          { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
40
```

Validates month value

Validates day value

Outline

Date.java

(2 of 3)

Outline

Date.java

```
41       // check if day in range for month
42       if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
43          return testDay;
44
45       // check for leap year
46       if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
47            ( year % 4 == 0 && year % 100 != 0 ) ) )
48          return testDay;
49
50       System.out.printf( "Invalid day (%d) set to 1.", testDay );
51       return 1;  // maintain object in consistent state
52    } // end method checkDay
53
54    // return a String of the form month/day/year
55    public String toString()
56    {
57       return String.format( "%d/%d/%d", month, day, year );
58    } // end method toString
59 } // end class Date
```

Check if the day is February 29 on a leap year

Outline

Employee.java

```
1  // Fig. 8.8: Employee.java
2  // Employee class with references to other objects.
3
4  public class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
10
11    // constructor to initialize name, birth date and hire date
12    public Employee( String first, String last, Date dateOfBirth,
13       Date dateOfHire )
14    {
15       firstName = first;
16       lastName = last;
17       birthDate = dateOfBirth;
18       hireDate = dateOfHire;
19    } // end Employee constructor
20
21    // convert Employee to String format
22    public String toString()
23    {
24       return String.format( "%s, %s  Hired: %s  Birthday: %s",
25          lastName, firstName, hireDate, birthDate );
26    } // end method toString
27 } // end class Employee
```

Employee contains references to two **Date** objects

Implicit calls to **hireDate** and **birthDate**'s **toString** methods

Outline

EmployeeTest.java

```
1  // Fig. 8.9: EmployeeTest.java
2  // Composition demonstration.
3
4  public class EmployeeTest
5  {
6     public static void main( String args[] )
7     {
8        Date birth = new Date( 7, 24, 1949 );
9        Date hire = new Date( 3, 12, 1988 );
10       Employee employee = new Employee( "Bob", "Blue", birth, hire );
11
12       System.out.println( employee );
13    } // end main
14 } // end class EmployeeTest
```

Create an **Employee** object

Display the **Employee** object

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

Enumerations

- **`enum` types**
  - **Declared with an `enum` declaration**
    - **A comma-separated list of `enum` constants**
    - **Declares an `enum` class with the following restrictions:**
      - **`enum` types are implicitly `final`**
      - **`enum` constants are implicitly `static`**
      - **Attempting to create an object of an `enum` type with `new` is a compilation error**
  - **`enum` constants can be used anywhere constants can**
  - **`enum` constructor**
    - **Like class constructors, can specify parameters and be overloaded**

Outline

Book.java

(1 of 2)

```
1  // Fig. 8.10: Book.java
2  // Declaring an enum type with constructor and explicit instance fields
3  // and accessors for these field
4
5  public enum Book
6  {
7     // declare constants of enum type
8     JHTP6( "Java How to Program 6e", "2005" ),
9     CHTP4( "C How to Program 4e", "2004" ),
10    IW3HTP3( "Internet & World Wide Web How to Program 3e", "2004" ),
11    CPPHTP4( "C++ How to Program 4e", "2003" ),
12    VBHTP2( "Visual Basic .NET How to Program 2e", "2002" ),
13    CSHARPHTP( "C# How to Program", "2002" );
14
15    // instance fields
16    private final String title; // book title
17    private final String copyrightYear; // copyright year
18
19    // enum constructor
20    Book( String bookTitle, String year )
21    {
22       title = bookTitle;
23       copyrightYear = year;
24    } // end enum Book constructor
25
```

Declare six **enum** constants

Arguments to pass to the **enum** constructor

Declare instance variables

Declare **enum** constructor **Book**

```
26    // accessor for field title
27    public String getTitle()
28    {
29       return title;
30    } // end method getTitle
31
32    // accessor for field copyrightYear
33    public String getCopyrightYear()
34    {
35       return copyrightYear;
36    } // end method getCopyrightYear
37 } // end enum Book
```

Outline

Book.java

(2 of 2)

Enumerations (Cont.)

- `static` method `values`
  - Generated by the compiler for every `enum`
  - Returns an array of the `enum`'s constants in the order in which they were declared
- `static` method `range` of class `EnumSet`
  - Takes two parameters, the first and last `enum` constants in the desired range
  - Returns an `EnumSet` containing the constants in that range, inclusive
  - An enhanced `for` statement can iterate over an `EnumSet` as it can over an array

Outline

EnumTest.java

(1 of 2)

```
1  // Fig. 8.11: EnumTest.java
2  // Testing enum type Book.
3  import java.util.EnumSet;
4
5  public class EnumTest
6  {
7     public static void main( String args[] )
8     {
9        System.out.println( "All books:\n" );
10
11       // print all books in enum Book
12       for ( Book book : Book.values() )
13          System.out.printf( "%-10s%-45s%s\n", book,
14             book.getTitle(), book.getCopyrightYear() );
15
16       System.out.println( "\nDisplay a range of enum constants:\n" );
17
18       // print first four books
19       for ( Book book : EnumSet.range( Book.JHTP6, Book.CPPHTP4 ) )
20          System.out.printf( "%-10s%-45s%s\n", book,
21             book.getTitle(), book.getCopyrightYear() );
22    } // end main
23 } // end class EnumTest
```

Enhanced **for** loop iterates for each **enum** constant in the array returned by method **value**

Enhanced **for** loop iterates for each **enum** constant in the **EnumSet** returned by method **range**

```
All books:

JHTP6     Java How to Program 6e                          2005
CHTP4     C How to Program 4e                             2004
IW3HTP3   Internet & World Wide Web How to Program 3e     2004
CPPHTP4   C++ How to Program 4e                           2003
VBHTP2    Visual Basic .NET How to Program 2e             2002
CSHARPHTP C# How to Program                               2002

Display a range of enum constants:

JHTP6     Java How to Program 6e                          2005
CHTP4     C How to Program 4e                             2004
IW3HTP3   Internet & World Wide Web How to Program 3e     2004
CPPHTP4   C++ How to Program 4e                           2003
```

Outline

EnumTest.java

(2 of 2)

Garbage Collection and Method `finalize`

- **Garbage collection**
  - **JVM marks an object for garbage collection when there are no more references to that object**
  - **JVM's garbage collector will retrieve those objects memory so it can be used for other objects**
- **`finalize` method**
  - **All classes in Java have the `finalize` method**
    - **Inherited from the `Object` class**
  - **`finalize` is called by the garbage collector when it performs termination housekeeping**
  - **`finalize` takes no parameters and has return type `void`**

`static` Class Members

- **`static` fields**
  - **Also known as class variables**
  - **Represents class-wide information**
  - **Used when:**
    - **all objects of the class should share the same copy of this instance variable or**
    - **this instance variable should be accessible even when no objects of the class exist**
  - **Can be accessed with the class name or an object name and a dot (.)**
  - **Must be initialized in their declarations, or else the compiler will initialize it with a default value (`0` for `int`s)**

Outline

Employee.java

(1 of 2)

```
1  // Fig. 8.12: Employee.java
2  // Static variable used to maintain a count of the number of
3  // Employee objects in memory.
4
5  public class Employee
6  {
7     private String firstName;
8     private String lastName;
9     private static int count = 0; // number of objects in memory
10
11    // initialize employee, add 1 to static count and
12    // output String indicating that constructor was called
13    public Employee( String first, String last )
14    {
15       firstName = first;
16       lastName = last;
17
18       count++;  // increment static count of employees
19       System.out.printf( "Employee constructor: %s %s; count = %d\n",
20          firstName, lastName, count );
21    } // end Employee constructor
22
```

Declare a **`static`** field

Increment **`static`** field

Outline

Employee.java

(2 of 2)

```
23    // subtract 1 from static count when garbage
24    // collector calls finalize to clean up object;
25    // confirm that finalize was called
26    protected void finalize()
27    {
28       count--; // decrement static count of employees
29       System.out.printf( "Employee finalizer: %s %s; count = %d\n",
30          firstName, lastName, count );
31    } // end method finalize
32
33    // get first name
34    public String getFirstName()
35    {
36       return firstName;
37    } // end method getFirstName
38
39    // get last name
40    public String getLastName()
41    {
42       return lastName;
43    } // end method getLastName
44
45    // static method to get static count value
46    public static int getCount()
47    {
48       return count;
49    } // end method getCount
50 } // end class Employee
```

Declare method **`finalize`**

Declare **`static`** method **`getCount`** to get **`static`** field **`count`**

Outline

EmployeeTest.java

(1 of 3)

```
1  // Fig. 8.13: EmployeeTest.java
2  // Static member demonstration.
3
4  public class EmployeeTest
5  {
6     public static void main( String args[] )
7     {
8        // show that count is 0 before creating Employees
9        System.out.printf( "Employees before instantiation: %d\n",
10          Employee.getCount() );
11
12       // create two Employees; count should be 2
13       Employee e1 = new Employee( "Susan", "Baker" );
14       Employee e2 = new Employee( "Bob", "Blue" );
15
```

Call **`static`** method **`getCount`** using class name **`Employee`**

Create new **`Employee`** objects

Outline

EmployeeTest.java

(2 of 3)

```
16      // show that count is 2 after creating two Employees
17      System.out.println( "\nEmployees after instantiation: " );
18      System.out.printf( "via e1.getCount(): %d\n", e1.getCount() );
19      System.out.printf( "via e2.getCount(): %d\n", e2.getCount() );
20      System.out.printf( "via Employee.getCount(): %d\n",
21         Employee.getCount() );
22
23      // get names of Employees
24      System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n\n",
25         e1.getFirstName(), e1.getLastName(),
26         e2.getFirstName(), e2.getLastName() );
27
28      // in this example, there is only one reference to each Employee,
29      // so the following two statements cause the JVM to mark each
30      // Employee object for garbage collection
31      e1 = null;
32      e2 = null;
33
34      System.gc(); // ask for garbage collection to occur now
35
```

Call **static** method **getCount** using class name

Call **static** method **getCount** using variable name

Remove references to objects, JVM will mark them for garbage collection

Call **static** method **gc** of class **System** to indicate that garbage collection should be attempted

Outline

EmployeeTest.java

(3 of 3)

```
36       // show Employee count after calling garbage collector; count
37       // displayed may be 0, 1 or 2 based on whether garbage collector
38       // executes immediately and number of Employee objects collected
39       System.out.printf( "\nEmployees after System.gc(): %d\n",
40          Employee.getCount() );
41    } // end main
42 } // end class EmployeeTest
```

Call **static** method **getCount**

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2

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

Employee 1: Susan Baker
Employee 2: Bob Blue

Employee finalizer: Bob Blue; count = 1
Employee finalizer: Susan Baker; count = 0

Employees after System.gc(): 0
```

`static` Class Members (Cont.)

- `String` objects are immutable
  - `String` concatenation operations actually result in the creation of a new `String` object
- `static` method `gc` of class `System`
  - Indicates that the garbage collector should make a best-effort attempt to reclaim objects eligible for garbage collection
  - It is possible that no objects or only a subset of eligible objects will be collected
- `static` methods cannot access non-`static` class members
  - Also cannot use the `this` reference

`static` Import

- **`static` import declarations**
  - **Enables programmers to refer to imported `static` members as if they were declared in the class that uses them**
  - **Single `static` import**
    - `import static packageName.ClassName.staticMemberName;`
  - **`static` import on demand**
    - `import static packageName.ClassName.*;`
    - **Imports all `static` members of the specified class**

Outline

StaticImportTest.java

```
1  // Fig. 8.14: StaticImportTest.java
2  // Using static import to import static methods of class Math.
3  import static java.lang.Math.*;
4
5  public class StaticImportTest
6  {
7     public static void main( String args[] )
8     {
9        System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
10       System.out.printf( "ceil( -9.8 ) = %.1f\n", ceil( -9.8 ) );
11       System.out.printf( "log( E ) = %.1f\n", log( E ) );
12       System.out.printf( "cos( 0.0 ) = %.1f\n", cos( 0.0 ) );
13    } // end main
14 } // end class StaticImportTest
```

static import on demand

Use **Math**'s **static** methods and instance variable without preceding them with **Math.**

```
sqrt( 900.0 ) = 30.0
ceil( -9.8 ) = -9.0
log( E ) = 1.0
cos( 0.0 ) = 1.0
```

`final` Instance Variables

- **Principle of least privilege**
  - **Code should have only the privilege and access it needs to accomplish its task, but no more**
- **`final` instance variables**
  - **Keyword `final`**
    - **Specifies that a variable is not modifiable (is a constant)**
  - **`final` instance variables can be initialized at their declaration**
    - **If they are not initialized in their declarations, they must be initialized in all constructors**

Outline

Increment.java

```
1  // Fig. 8.15: Increment.java
2  // final instance variable in a class.
3
4  public class Increment
5  {
6     private int total = 0; // total of all increments
7     private final int INCREMENT; // constant variable (uninitialized)
8
9     // constructor initializes final instance variable INCREMENT
10    public Increment( int incrementValue )
11    {
12       INCREMENT = incrementValue; // initialize constant variable (once)
13    } // end Increment constructor
14
15    // add INCREMENT to total
16    public void addIncrementToTotal()
17    {
18       total += INCREMENT;
19    } // end method addIncrementToTotal
20
21    // return String representation of an Increment object's data
22    public String toString()
23    {
24       return String.format( "total = %d", total );
25    } // end method toIncrementString
26 } // end class Increment
```

Declare **`final`** instance variable

Initialize **`final`** instance variable inside a constructor

Outline

IncrementTest.java

```
1  // Fig. 8.16: IncrementTest.java
2  // final variable initialized with a constructor argument.
3
4  public class IncrementTest
5  {
6     public static void main( String args[] )
7     {
8        Increment value = new Increment( 5 );
9
10       System.out.printf( "Before incrementing: %s\n\n", value );
11
12       for ( int i = 1; i <= 3; i++ )
13       {
14          value.addIncrementToTotal();
15          System.out.printf( "After increment %d: %s\n", i, value );
16       } // end for
17    } // end main
18 } // end class IncrementTest
```

Create an **Increment** object

Call method **addIncrementToTotal**

```
Before incrementing: total = 0

After increment 1: total = 5
After increment 2: total = 10
After increment 3: total = 15
```

Software Reusability

- **Rapid application development**
  - **Software reusability speeds the development of powerful, high-quality software**
- **Java's API**
  - **provides an entire framework in which Java developers can work to achieve true reusability and rapid application development**
  - **Documentation:**
    - **java.sun.com/javase/6/docs/api/**
    - **Or http://java.sun.com/javase/downloads/index.jsp to download**

Data Abstraction and Encapsulation

- **Data abstraction**
  - **Information hiding**
    - **Classes normally hide the details of their implementation from their clients**
  - **Abstract data types (ADTs)**
    - **Data representation**
      - **example: primitive type `int` is an abstract representation of an integer**
        - **`int`s are only approximations of integers, can produce arithmetic overflow**
    - **Operations that can be performed on data**

Data Abstraction and Encapsulation (Cont.)

- **Queues**
  - **Similar to a "waiting line"**
    - **Clients place items in the queue (enqueue an item)**
    - **Clients get items back from the queue (dequeue an item)**
    - **First-in, first out (FIFO) order**
  - **Internal data representation is hidden**
    - **Clients only see the ability to enqueue and dequeue items**

Time Class Case Study: Creating Packages

- To declare a reusable class
  - Declare a `public` class
  - Add a `package` declaration to the source-code file
    - must be the first executable statement in the file
    - `package` name should consist of your Internet domain name in reverse order followed by other names for the package
      - example: `com.deitel.jhtp7.ch08`
      - `package` name is part of the fully qualified class name
        - Distinguishes between multiple classes with the same name belonging to different packages
        - Prevents name conflict (also called name collision)
      - Class name without `package` name is the simple name

Outline

```
1  // Fig. 8.18: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3  package com.deitel.jhtp7.ch08;
4
5  public class Time1
6  {
7     private int hour;   // 0 - 23
8     private int minute; // 0 - 59
9     private int second; // 0 - 59
10
11    // set a new time value using universal time; perform
12    // validity checks on the data; set invalid values to zero
13    public void setTime( int h, int m, int s )
14    {
15       hour = ( ( h >= 0 && h < 24 ) ? h : 0 );   // validate hour
16       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
17       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
18    } // end method setTime
19
```

`package` declaration

`Time1` is a **`public`** class so it can be used by importers of this package

`Time1.java`

Outline

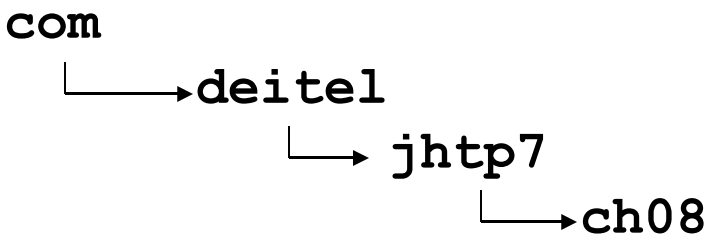
Time1.java

(2 of 2)

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

Time Class Case Study: Creating Packages (Cont.)

- Compile the class so that it is placed in the appropriate package directory structure
  - Example: our package should be in the directory



  - `javac` command-line option `-d`
    - `javac` creates appropriate directories based on the class's `package` declaration
    - A period (`.`) after `-d` represents the current directory

Time Class Case Study: Creating Packages (Cont.)

- Import the reusable class into a program
  - Single-type-import declaration
    - Imports a single class
    - Example: `import java.util.Random;`
  - Type-import-on-demand declaration
    - Imports all classes in a package
    - Example: `import java.util.*;`

Outline

```
1  // Fig. 8.19: Time1PackageTest.java
2  // Time1 object used in an application.
3  import com.deitel.jhtp6.ch08.Time1; // import class Time1
4
5  public class Time1PackageTest
6  {
7     public static void main( String args[] )
8     {
9        // create and initialize a Time1 object
10       Time1 time = new Time1(); // calls Time1 constructor
11
12       // output string representations of the time
13       System.out.print( "The initial universal time is: " );
14       System.out.println( time.toUniversalString() );
15       System.out.print( "The initial standard time is: " );
16       System.out.println( time.toString() );
17       System.out.println(); // output a blank line
18
```

Single-type **`import`** declaration

Refer to the **`Time1`** class by its simple name

`Time1PackageTest.java`

(1 of 2)

Outline

Time1PackageTest.java

(2 of 2)

```
19       // change time and output updated time
20       time.setTime( 13, 27, 6 );
21       System.out.print( "Universal time after setTime is: " );
22       System.out.println( time.toUniversalString() );
23       System.out.print( "Standard time after setTime is: " );
24       System.out.println( time.toString() );
25       System.out.println(); // output a blank line
26
27       // set time with invalid values; output updated time
28       time.setTime( 99, 99, 99 );
29       System.out.println( "After attempting invalid settings:" );
30       System.out.print( "Universal time: " );
31       System.out.println( time.toUniversalString() );
32       System.out.print( "Standard time: " );
33       System.out.println( time.toString() );
34    } // end main
35 } // end class Time1PackageTest
```

```
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
```

Time Class Case Study: Creating Packages (Cont.)

- **Class loader**
  - **Locates classes that the compiler needs**
    - **First searches standard Java classes bundled with the JDK**
    - **Then searches for optional packages**
      - **These are enabled by Java's extension mechanism**
    - **Finally searches the classpath**
      - **List of directories or archive files separated by directory separators**
        - **These files normally end with `.jar` or `.zip`**
        - **Standard classes are in the archive file `rt.jar`**

Time Class Case Study: Creating Packages (Cont.)

- To use a classpath other than the current directory
  - `-classpath` option for the `javac` compiler
  - Set the `CLASSPATH` environment variable
- The JVM must locate classes just as the compiler does
  - The `java` command can use other classpathes by using the same techniques that the `javac` command uses

Common Programming Error 8.13

Specifying an explicit classpath eliminates the current directory from the classpath. This prevents classes in the current directory (including packages in the current directory) from loading properly. If classes must be loaded from the current directory, include a dot (.) in the classpath to specify the current directory.

Software Engineering Observation 8.16

In general, it is a better practice to use the `-classpath` option of the compiler, rather than the `CLASSPATH` environment variable, to specify the classpath for a program. This enables each application to have its own classpath.

Package Access

- **Package access**
  - **Methods and variables declared without any access modifier are given package access**
  - **This has no effect if the program consists of one class**
  - **This does have an effect if the program contains multiple classes from the same package**
    - **Package-access members can be directly accessed through the appropriate references to objects in other classes belonging to the same package**

Outline

PackageDataTest.java

(1 of 2)

```
1  // Fig. 8.20: PackageDataTest.java
2  // Package-access members of a class are accessible by other classes
3  // in the same package.
4
5  public class PackageDataTest
6  {
7     public static void main( String args[] )
8     {
9        PackageData packageData = new PackageData();
10
11       // output String representation of packageData
12       System.out.printf( "After instantiation:\n%s\n", packageData );
13
14       // change package access data in packageData object
15       packageData.number = 77;
16       packageData.string = "Goodbye";
17
18       // output String representation of packageData
19       System.out.printf( "\nAfter changing values:\n%s\n", packageData );
20    } // end main
21 } // end class PackageDataTest
22
```

Can directly access package-access members

Outline

PackageDataTest.java

(2 of 2)

```
23 // class with package access instance variables
24 class PackageData
25 {
26    int number; // package-access instance variable
27    String string; // package-access instance variable
28
29    // constructor
30    public PackageData()
31    {
32       number = 0;
33       string = "Hello";
34    } // end PackageData constructor
35
36    // return PackageData object String representation
37    public String toString()
38    {
39       return String.format( "number: %d; string: %s", number, string );
40    } // end method toString
41 } // end class PackageData
```

Package-access instance variables

```
After instantiation:
number: 0; string: Hello

After changing values:
number: 77; string: Goodbye
```

Object-Oriented Programming: Inheritance

WE WILL COVER

- How inheritance promotes software reusability.
- The notions of superclasses and subclasses.
- To use keyword `extends` to create a class that inherits attributes and behaviors from another class.
- To use access modifier `protected` to give subclass methods access to superclass members.
- To access superclass members with `super`.
- How constructors are used in inheritance hierarchies.
- The methods of class `Object`, the direct or indirect superclass of all classes in Java.

Outline

Outline

Introduction

- **Inheritance**
  - **Software reusability**
  - **Create new class from existing class**
    - **Absorb existing class's data and behaviors**
    - **Enhance with new capabilities**
  - **Subclass extends superclass**
    - **Subclass**
      - **More specialized group of objects**
      - **Behaviors inherited from superclass**
        - **Can customize**
      - **Additional behaviors**

Introduction (Cont.)

- **Class hierarchy**
  - **Direct superclass**
    - **Inherited explicitly (one level up hierarchy)**
  - **Indirect superclass**
    - **Inherited two or more levels up hierarchy**
  - **Single inheritance**
    - **Inherits from one superclass**
  - **Multiple inheritance**
    - **Inherits from multiple superclasses**
      - **Java does not support multiple inheritance**

Superclasses and subclasses

- **Superclasses and subclasses**
  - **Object of one class "is an" object of another class**
    - **Example: Rectangle is quadrilateral.**
      - **Class `Rectangle` inherits from class `Quadrilateral`**
      - **`Quadrilateral`: superclass**
      - **`Rectangle`: subclass**
  - **Superclass typically represents larger set of objects than subclasses**
    - **Example:**
      - **superclass: `Vehicle`**
        - **Cars, trucks, boats, bicycles, …**
      - **subclass: `Car`**
        - **Smaller, more-specific subset of vehicles**

Superclass	Subclasses
`Student`	`GraduateStudent`, `UndergraduateStudent`
`Shape`	`Circle`, `Triangle`, `Rectangle`
`Loan`	`CarLoan`, `HomeImprovementLoan`, `MortgageLoan`
`Employee`	`Faculty`, `Staff`
`BankAccount`	`CheckingAccount`, `SavingsAccount`

Fig. 9.1 | Inheritance examples.

Superclasses and subclasses (Cont.)

- **Inheritance hierarchy**
  - **Inheritance relationships: tree-like hierarchy structure**
  - **Each class becomes**
    - **superclass**
      - **Supply members to other classes**

    OR

    - **subclass**
      - **Inherit members from other classes**

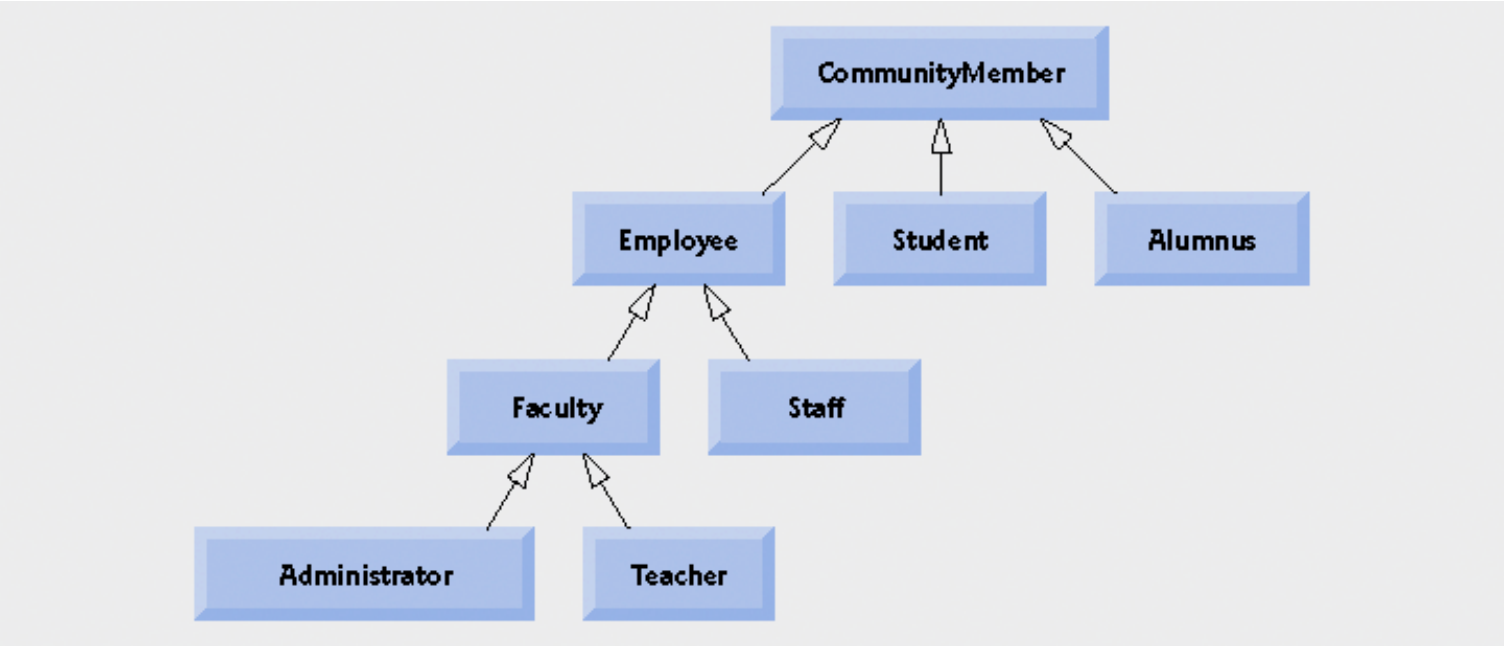


Fig. 9.2 | Inheritance hierarchy for university `CommunityMembers`

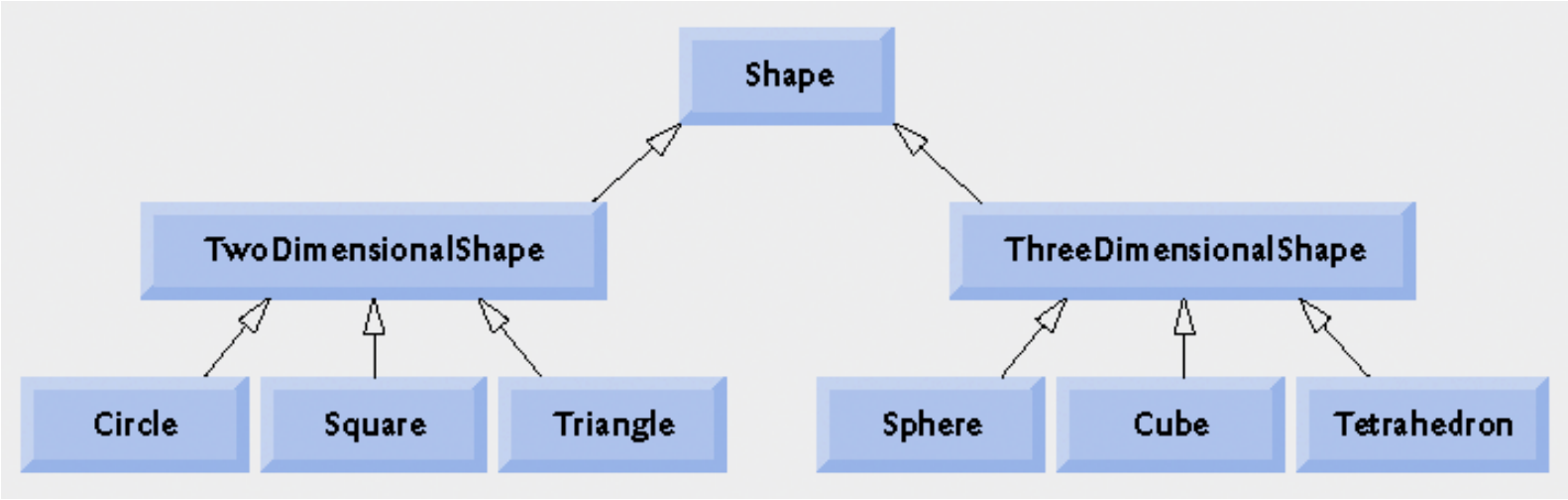


Fig. 9.3 | Inheritance hierarchy for Shapes.

protected Members

- protected access
  - Intermediate level of protection between public and private
  - protected members accessible by
    - superclass members
    - subclass members
    - Class members in the same package
  - Subclass access to superclass member
    - Keyword super and a dot (.)

Software Engineering Observation 9.1

Methods of a subclass cannot directly access `private` members of their superclass. A subclass can change the state of `private` superclass instance variables only through non-`private` methods provided in the superclass and inherited by the subclass.

Relationship between Superclasses and Subclasses

- **Superclass and subclass relationship**
  - **Example: `CommissionEmployee/BasePlusCommissionEmployee` inheritance hierarchy**
    - `CommissionEmployee`
      - **First name, last name, SSN, commission rate, gross sale amount**
    - `BasePlusCommissionEmployee`
      - **First name, last name, SSN, commission rate, gross sale amount**
      - **Base salary**

Creating and Using a `CommissionEmployee` Class

- Class `CommissionEmployee`
  - Extends class `Object`
    - Keyword `extends`
    - Every class in Java extends an existing class
      - Except `Object`
    - Every class inherits `Object`'s methods
    - New class implicitly extends Object
      - If it does not extend another class

```
1  // Fig. 9.4: CommissionEmployee.java
2  // CommissionEmployee class represents a commission empl
3
4  public class CommissionEmployee extends Object
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percen
11
12    // five-argument constructor
13    public CommissionEmployee( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // implicit call to Object constructor occurs h
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // validate and store gross sales
21       setCommissionRate( rate ); // validate and store commission rate
22    } // end five-argument CommissionEmployee constructor
23
24    // set first name
25    public void setFirstName( String first )
26    {
27       firstName = first;
28    } // end method setFirstName
29
```

Declare `private` instance variables

Class `CommissionEmployee` `extends` class `Object`

Implicit call to `Object` constructor

Initialize instance variables

Invoke methods `setGrossSales` and `setCommissionRate` to validate data

Outline

`issionEmployee`
`.java`

(1 of 4)

Line 4

Lines 6-10

Line 16

Lines 20-21

Outline

CommissionEmployee.java

(2 of 4)

```
30    // return first name
31    public String getFirstName()
32    {
33       return firstName;
34    } // end method getFirstName
35
36    // set last name
37    public void setLastName( String last )
38    {
39       lastName = last;
40    } // end method setLastName
41
42    // return last name
43    public String getLastName()
44    {
45       return lastName;
46    } // end method getLastName
47
48    // set social security number
49    public void setSocialSecurityNumber( String ssn )
50    {
51       socialSecurityNumber = ssn; // should validate
52    } // end method setSocialSecurityNumber
53
54    // return social security number
55    public String getSocialSecurityNumber()
56    {
57       return socialSecurityNumber;
58    } // end method getSocialSecurityNumber
59
```

Outline

CommissionEmployee.java

(3 of 4)

Lines 85-88

```
60    // set gross sales amount
61    public void setGrossSales( double sales )
62    {
63       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64    } // end method setGrossSales
65
66    // return gross sales amount
67    public double getGrossSales()
68    {
69       return grossSales;
70    } // end method getGrossSales
71
72    // set commission rate
73    public void setCommissionRate( double rate )
74    {
75       commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76    } // end method setCommissionRate
77
78    // return commission rate
79    public double getCommissionRate()
80    {
81       return commissionRate;
82    } // end method getCommissionRate
83
84    // calculate earnings
85    public double earnings()
86    {
87       return commissionRate * grossSales;
88    } // end method earnings
89
```

Calculate earnings

```
90    // return String representation of CommissionEmployee object
91    public String toString()
92    {
93       return String.format( "%s: %s %s\n%s: %s\n%s: %
94          "commission employee", firstName, lastName,
95          "social security number", socialSecurityNumber,
96          "gross sales", grossSales,
97          "commission rate", commissionRate );
98    } // end method toString
99 } // end class CommissionEmployee
```

Override method `toString` of class `Object`

Outline

CommissionEmployee.java

(4 of 4)

Lines 91-98

Outline

```
1  // Fig. 9.5: CommissionEmployeeTest.java
2  // Testing class CommissionEmployee.
3
4  public class CommissionEmployeeTest
5  {
6     public static void main( String args[] )
7     {
8        // instantiate CommissionEmployee object
9        CommissionEmployee employee = new CommissionEmployee(
10          "Sue", "Jones", "222-22-2222", 10000, .06 );
11
12       // get commission employee data
13       System.out.println(
14          "Employee information obtained by get methods: \n" );
15       System.out.printf( "%s %s\n", "
16          employee.getFirstName() );
17       System.out.printf( "%s %s\n", "
18          employee.getLastName() );
19       System.out.printf( "%s %s\n", "Social security number is",
20          employee.getSocialSecurityNumber() );
21       System.out.printf( "%s %.2f\n", "Gross sales is"
22          employee.getGrossSales() );
23       System.out.printf( "%s %.2f\n", "Commiss
24          employee.getCommissionRate() );
25
26       employee.setGrossSales( 500 ); // set gross sales
27       employee.setCommissionRate( .1 ); // set commission rate
28
```

Instantiate `CommissionEmployee` object

Use `CommissionEmployee`'s *get* methods to retrieve the object's instance variable values

Use `CommissionEmployee`'s *set* methods to change the object's instance variable values

CommissionEmployeeTest.java

(1 of 2)

Lines 9-10

Lines 15-25

26-27

```
29       System.out.printf( "\n%s:\n\n%s\n",
30          "Updated employee information obtained by toString", employee );
31    } // end main
32 } // end class CommissionEmployeeTest
```

```
Employee information obtained by get methods:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information obtained by toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 500.00
commission rate: 0.10
```

Implicitly call object's `toString` method

Outline

`onEmployee`
`Test.java`

(2 of 2)

Line 30

Program output

Creating a BasePlusCommissionEmployee Class without Using Inheritance

- Class **BasePlusCommissionEmployee**
  - Implicitly extends **Object**
  - Much of the code is similar to **CommissionEmployee**
    - **private** instance variables
    - **public** methods
    - constructor
  - Additions
    - **private** instance variable **baseSalary**
    - Methods **setBaseSalary** and **getBaseSalary**

Outline

BasePlusCommissionEmployee.java

```
1  // Fig. 9.6: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class represents an employee that receives
3  // a base salary in addition to commission.
4
5  public class BasePlusCommissionEmployee
6  {
7     private String firstName;
8     private String lastName;
9     private String socialSecurityNumber;
10    private double grossSales; // gross weekly sales
11    private double commissionRate; // commission percentage
12    private double baseSalary; // base salary per week
13
14    // six-argument constructor
15    public BasePlusCommissionEmployee( String first, String last,
16       String ssn, double sales, double rate, double salary )
17    {
18       // implicit call to Object constructor occurs here
19       firstName = first;
20       lastName = last;
21       socialSecurityNumber = ssn;
22       setGrossSales( sales ); // validate and store
23       setCommissionRate( rate ); // validate and store commission rate
24       setBaseSalary( salary ); // validate and store base salary
25    } // end six-argument BasePlusCommissionEmployee constructor
26
```

Add instance variable `baseSalary`

Line 12

Line 24

Use method `setBaseSalary` to validate data

```
27    // set first name
28    public void setFirstName( String first )
29    {
30       firstName = first;
31    } // end method setFirstName
32
33    // return first name
34    public String getFirstName()
35    {
36       return firstName;
37    } // end method getFirstName
38
39    // set last name
40    public void setLastName( String last )
41    {
42       lastName = last;
43    } // end method setLastName
44
45    // return last name
46    public String getLastName()
47    {
48       return lastName;
49    } // end method getLastName
50
51    // set social security number
52    public void setSocialSecurityNumber( String ssn )
53    {
54       socialSecurityNumber = ssn; // should validate
55    } // end method setSocialSecurityNumber
56
```

Outline

BasePlusCommissionEmployee.java

(2 of 4)

Outline

BasePlusCommissionEmployee.java

(3 of 4)

```
57    // return social security number
58    public String getSocialSecurityNumber()
59    {
60       return socialSecurityNumber;
61    } // end method getSocialSecurityNumber
62
63    // set gross sales amount
64    public void setGrossSales( double sales )
65    {
66       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
67    } // end method setGrossSales
68
69    // return gross sales amount
70    public double getGrossSales()
71    {
72       return grossSales;
73    } // end method getGrossSales
74
75    // set commission rate
76    public void setCommissionRate( double rate )
77    {
78       commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
79    } // end method setCommissionRate
80
81    // return commission rate
82    public double getCommissionRate()
83    {
84       return commissionRate;
85    } // end method getCommissionRate
86
```

Outline

BasePlusCommissionEmployee.java

(4 of 4)

Lines 88-91

Lines 94-97

Line 102

Lines 108-113

```
87    // set base salary
88    public void setBaseSalary( double salary )
89    {
90       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
91    } // end method setBaseSalary
92
93    // return base salary
94    public double getBaseSalary()
95    {
96       return baseSalary;
97    } // end method getBaseSalary
98
99    // calculate ea
100      public double
101      {
102         return baseSalary + ( commissionRate * grossSales );
103      } // end method earnings
104
105      // return String representation of BasePlusCommissionEmployee
106      public String toString()
107      {
108         return String.format(
109            "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
110            "base-salaried commission employee", firstName, lastName,
111            "social security number", socialSecurityNumber,
112            "gross sales", grossSales, "commission rate"
113            "base salary", baseSalary );
114      } // end method toString
115   } // end class BasePlusCommissionEmployee
```

Method `setBaseSalary` validates data and sets instance variable `baseSalary`

Method `getBaseSalary` returns the value of instance variable `baseSalary`

Update method `earnings` to calculate the earnings of a base-salaried commission employee

Update method `toString` to display base salary

Outline

```
1  // Fig. 9.7: BasePlusCommissionEmployeeTest.java
2  // Testing class BasePlusCommissionEmployee.
3
4  public class BasePlusCommissionEmployeeTest
5  {
6     public static void main( String args[] )
7     {
8        // instantiate BasePlusCommissionEmployee object
9        BasePlusCommissionEmployee employee =
10          new BasePlusCommissionEmployee(
11          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13       // get base-salaried commission employee data
14       System.out.println(
15          "Employee information obtained by get methods: \n" );
16       System.out.printf( "%s %s\n",
17          employee.getFirstName() );
18       System.out.printf( "%s %s\n",
19          employee.getLastName() );
20       System.out.printf( "%s %s\n", "Social security number is",
21          employee.getSocialSecurityNumber() );
22       System.out.printf( "%s %.2f\n", "Gross sales is",
23          employee.getGrossSales() );
24       System.out.printf( "%s %.2f\n", "Commission rate is",
25          employee.getCommissionRate() );
26       System.out.printf( "%s %.2f\n", "Base salary is",
27          employee.getBaseSalary() );
28
```

sePlusCommission
ployeeTest.java

(1 of 2)

Instantiate `BasePlusCommissionEmployee` object

Line 9-11

Use `BasePluCommissionEmployee`'s *get* methods to retrieve the object's instance variable values

Lines 16-27

```
29       employee.setBaseSalary( 1000 ); // set base salary
30
31       System.out.printf( "\n%s:\n\n%s\n",
32          "Updated employee information obta
33           employee.toString() );
34    } // end main
35 } // end class BasePlusCommissionEmplo
```

```
Employee information obtained by get met

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Use `BasePlusCommissionEmployee`'s `setBaseSalary` methods to set base salary

Explicitly call object's `toString` method

Outline

`BasePlusCommissionEmployeeTest.java`

(2 of 2)

Line 29

Line 33

Program output

Software Engineering Observation 9.4

Copying and pasting code from one class to another can spread errors across multiple source code files. To avoid duplicating code (and possibly errors), use inheritance, rather than the "copy-and-paste" approach, in situations where you want one class to "absorb" the instance variables and methods of another class.

Software Engineering Observation 9.5

With inheritance, the common instance variables and methods of all the classes in the hierarchy are declared in a superclass. When changes are required for these common features, software developers need only to make the changes in the superclass—subclasses then inherit the changes. Without inheritance, changes would need to be made to all the source code files that contain a copy of the code in question.

Creating a CommissionEmployee-BasePlusCommiionEmployee Inheritance Hierarchy

- Class `BasePlusCommissionEmployee2`
  - Extends class `CommissionEmployee`
  - Is a `CommissionEmployee`
  - Has instance variable `baseSalary`
  - Inherits `public` and `protected` members
  - Constructor not inherited

Outline

BasePlusCommission
ee2.java

Line 4

Line 13

```
1  // Fig. 9.8: BasePlusCommissionEmployee2.java
2  // BasePlusCommissionEmployee2 inherits from class CommissionEmployee.
3
4  public class BasePlusCommissionEmployee2 extends CommissionEmployee
5  {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee2( String first, String last,
10       String ssn, double sales, double rate, double salary )
11    {
12       // explicit call to superclass CommissionEmployee constructor
13       super( first, last, ssn, sales, rate );
14
15       setBaseSalary( amount ); // validate and store base salary
16    } // end six-argument BasePlusCommissi
17
18    // set base salary
19    public void setBaseSalary( double salary )
20    {
21       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
22    } // end method setBaseSalary
23
```

Class `BasePluCommissionEmployee2` is a subclass of `CommissionEmployee`

Invoke the superclass constructor using the superclass constructor call syntax

Outline

BasePlusCommission

```
24    // return base salary
25    public double getBaseSalary()
26    {
27       return baseSalary;
28    } // end method getBaseSalary
29
30    // calculate earnings
31    public double earnings()
32    {
33       // not allowed: commissionRate and grossSales private in superclass
34       return baseSalary + ( commissionRate * grossSales );
35    } // end method earnings
36
37    // return String representation
38    public String toString()
39    {
40       // not allowed: attempts to a
41       return String.format(
42          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
43          "base-salaried commission employee", firstName, lastName,
44          "social security number", socialSecurityNumber,
45          "gross sales", grossSales, "commission rate", commissionRate,
46          "base salary", baseSalary );
47    } // end method toString
48 } // end class BasePlusCommissionEmployee2
```

Compiler generates errors because superclass's instance variable `commissionRate` and `grossSales` are `private`

Line 34

Lines 41-46

Compiler generates errors because superclass's instance variable `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` are `private`

```
BasePlusCommissionEmployee2.java:34: commissionRate has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                            ^
BasePlusCommissionEmployee2.java:34: grossSales has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                                             ^
BasePlusCommissionEmployee2.java:43: firstName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                              ^
BasePlusCommissionEmployee2.java:43: lastName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                                         ^
BasePlusCommissionEmployee2.java:44: socialSecurityNumber has private access in
CommissionEmployee
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee2.java:45: grossSales has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
BasePlusCommissionEmployee2.java:45: commissionRate has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                                                      ^
7 errors
```

Outline

BasePlusCommissionEmployee2.java

(3 of 3)

Compiler generated errorss

CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables

- Use `protected` instance variables
  - Enable class `BasePlusCommissionEmployee` to directly access superclass instance variables
  - Superclass's `protected` members are inherited by all subclases of that superclass

```
1  // Fig. 9.9: CommissionEmployee2.java
2  // CommissionEmployee2 class represents a commission employee.
3
4  public class CommissionEmployee2
5  {
6     protected String firstName;
7     protected String lastName;
8     protected String socialSecurityNumber;
9     protected double grossSales; // gross weekly sales
10    protected double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee2( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // implicit call to Object constructor occurs here
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // validate and store gross sales
21       setCommissionRate( rate ); // validate and store commission rate
22    } // end five-argument CommissionEmployee2 constructor
23
24    // set first name
25    public void setFirstName( String first )
26    {
27       firstName = first;
28    } // end method setFirstName
29
```

Declare `protected` instance variables

Outline

CommissionEmployee2.java

(1 of 4)

Line 6-10

```
30    // return first name
31    public String getFirstName()
32    {
33       return firstName;
34    } // end method getFirstName
35
36    // set last name
37    public void setLastName( String last )
38    {
39       lastName = last;
40    } // end method setLastName
41
42    // return last name
43    public String getLastName()
44    {
45       return lastName;
46    } // end method getLastName
47
48    // set social security number
49    public void setSocialSecurityNumber( String ssn )
50    {
51       socialSecurityNumber = ssn; // should validate
52    } // end method setSocialSecurityNumber
53
54    // return social security number
55    public String getSocialSecurityNumber()
56    {
57       return socialSecurityNumber;
58    } // end method getSocialSecurityNumber
59
```

Outline

Commission

Employee2.java

(2 of 4)

Outline

Commission

Employee2.java

(3 of 4)

```
60    // set gross sales amount
61    public void setGrossSales( double sales )
62    {
63       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64    } // end method setGrossSales
65
66    // return gross sales amount
67    public double getGrossSales()
68    {
69       return grossSales;
70    } // end method getGrossSales
71
72    // set commission rate
73    public void setCommissionRate( double rate )
74    {
75       commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76    } // end method setCommissionRate
77
78    // return commission rate
79    public double getCommissionRate()
80    {
81       return commissionRate;
82    } // end method getCommissionRate
83
84    // calculate earnings
85    public double earnings()
86    {
87       return commissionRate * grossSales;
88    } // end method earnings
89
```

```
90    // return String representation of CommissionEmployee2 object
91    public String toString()
92    {
93       return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
94          "commission employee", firstName, lastName,
95          "social security number", socialSecurityNumber,
96          "gross sales", grossSales,
97          "commission rate", commissionRate );
98    } // end method toString
99 } // end class CommissionEmployee2
```

Outline

Commission Employee2.java

(4 of 4)

Outline

BasePlusCommissionEmployee3.java

(1 of 2)

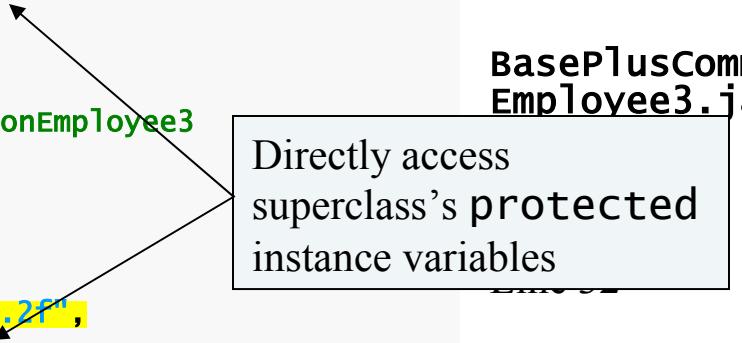
```
1  // Fig. 9.10: BasePlusCommissionEmployee3.java
2  // BasePlusCommissionEmployee3 inherits from CommissionEmployee2 and has
3  // access to CommissionEmployee2's protected members.
4
5  public class BasePlusCommissionEmployee3 extends CommissionEmployee2
6  {
7     private double baseSalary; // base salary per week
8
9     // six-argument constructor
10    public BasePlusCommissionEmployee3( String first, Strin
11       String ssn, double sales, double rate, double salary
12    {
13       super( first, last, ssn, sales, rate );
14       setBaseSalary( salary ); // validate and store base salary
15    } // end six-argument BasePlusCommissionEmployee3 constructor
16
17    // set base salary
18    public void setBaseSalary( double salary )
19    {
20       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
21    } // end method setBaseSalary
22
23    // return base salary
24    public double getBaseSalary()
25    {
26       return baseSalary;
27    } // end method getBaseSalary
28
```

Must call superclass's constructor

ne 13

Outline

```
29    // calculate earnings
30    public double earnings()
31    {
32       return baseSalary + ( commissionRate * grossSales );
33    } // end method earnings
34
35    // return String representation of BasePlusCommissionEmployee3
36    public String toString()
37    {
38       return String.format(
39          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
40          "base-salaried commission employee", firstName, lastName,
41          "social security number", socialSecurityNumber,
42          "gross sales", grossSales, "commission rate", commissionRate,
43          "base salary", baseSalary );
44    } // end method toString
45 } // end class BasePlusCommissionEmployee3
```



BasePlusCommissionEmployee3.java

Lines 38-43

Outline

BasePlusCommissionEmployeeTest3.java

(1 of 2)

```
1  // Fig. 9.11: BasePlusCommissionEmployeeTest3.java
2  // Testing class BasePlusCommissionEmployee3.
3
4  public class BasePlusCommissionEmployeeTest3
5  {
6     public static void main( String args[] )
7     {
8        // instantiate BasePlusCommissionEmployee3 object
9        BasePlusCommissionEmployee3 employee =
10          new BasePlusCommissionEmployee3(
11          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13       // get base-salaried commission employee data
14       System.out.println(
15          "Employee information obtained by get methods: \n" );
16       System.out.printf( "%s %s\n", "First name is",
17          employee.getFirstName() );
18       System.out.printf( "%s %s\n", "Last name is",
19          employee.getLastName() );
20       System.out.printf( "%s %s\n", "Social security number is",
21          employee.getSocialSecurityNumber() );
22       System.out.printf( "%s %.2f\n", "Gross sales is",
23          employee.getGrossSales() );
24       System.out.printf( "%s %.2f\n", "Commission rate is",
25          employee.getCommissionRate() );
26       System.out.printf( "%s %.2f\n", "Base salary is",
27          employee.getBaseSalary() );
28
```

Outline

```
29       employee.setBaseSalary( 1000 ); // set base salary
30
31       System.out.printf( "\n%s:\n\n%s\n",
32          "Updated employee information obtained by toString",
33          employee.toString() );
34    } // end main
35 } // end class BasePlusCommissionEmployeeTest3
```

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

BasePlusCommissionEmployeeTest3.java

(2 of 2)

Program output

CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- Using `protected` instance variables
  - Advantages
    - subclasses can modify values directly
    - Slight increase in performance
      - Avoid set/get method call overhead
  - Disadvantages
    - No validity checking
      - subclass can assign illegal value
    - Implementation dependent
      - subclass methods more likely dependent on superclass implementation
      - superclass implementation changes may result in subclass modifications
        - Fragile (brittle) software

Software Engineering Observation 9.6

Use the `protected` access modifier when a superclass should provide a method only to its subclasses and other classes in the same package, but not to other clients.

CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Uing private Instance Variables

- **Reexamine hierarchy**
  - **Use the best software engineering practice**
    - **Declare instance variables as `private`**
    - **Provide public *get* and *set* methods**
    - **Use *get* method to obtain values of instance variables**

```
1  // Fig. 9.12: CommissionEmployee3.java
2  // CommissionEmployee3 class represents a commission employee.
3
4  public class CommissionEmployee3
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee3( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // implicit call to Object constructor occurs here
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // validate and store gross sales
21       setCommissionRate( rate ); // validate and store commission rate
22    } // end five-argument CommissionEmployee3 constructor
23
24    // set first name
25    public void setFirstName( String first )
26    {
27       firstName = first;
28    } // end method setFirstName
29
```

Declare `private` instance variables

Outline

CommissionEmployee3.java

(1 of 4)

Lines 6-10

```
30    // return first name
31    public String getFirstName()
32    {
33       return firstName;
34    } // end method getFirstName
35
36    // set last name
37    public void setLastName( String last )
38    {
39       lastName = last;
40    } // end method setLastName
41
42    // return last name
43    public String getLastName()
44    {
45       return lastName;
46    } // end method getLastName
47
48    // set social security number
49    public void setSocialSecurityNumber( String ssn )
50    {
51       socialSecurityNumber = ssn; // should validate
52    } // end method setSocialSecurityNumber
53
54    // return social security number
55    public String getSocialSecurityNumber()
56    {
57       return socialSecurityNumber;
58    } // end method getSocialSecurityNumber
59
```

Outline

Commission

Employee3.java

(2 of 4)

Outline

Commission

Employee3.java

(3 of 4)

```
60    // set gross sales amount
61    public void setGrossSales( double sales )
62    {
63       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64    } // end method setGrossSales
65
66    // return gross sales amount
67    public double getGrossSales()
68    {
69       return grossSales;
70    } // end method getGrossSales
71
72    // set commission rate
73    public void setCommissionRate( double rate )
74    {
75       commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76    } // end method setCommissionRate
77
78    // return commission rate
79    public double getCommissionRate()
80    {
81       return commissionRate;
82    } // end method getCommissionRate
83
```

```
84    // calculate earnings
85    public double earnings()
86    {
87       return getCommissionRate() * getGrossSales();
88    } // end method earnings
89
90    // return String representation of CommissionEmployee
91    public String toString()
92    {
93       return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
94          "commission employee", getFirstName(), getLastName(),
95          "social security number", getSocialSecurityNumber(),
96          "gross sales", getGrossSales(),
97          "commission rate", getCommissionRate() );
98    } // end method toString
99 } // end class CommissionEmployee3
```

Use *get* methods to obtain the values of instance variables

Outline

ission

Employee3.java

(4 of 4)

Line 87

Lines 94-97

Outline

BasePlusCommission
Employee4.java

```
1  // Fig. 9.13: BasePlusCommissionEmployee4.java
2  // BasePlusCommissionEmployee4 class inherits from CommissionEmployee3 and
3  // accesses CommissionEmployee3's private data via CommissionEmployee3's
4  // public methods.
5
6  public class BasePlusCommissionEmployee4 extends CommissionEmployee3
7  {
8     private double baseSalary; // base salary per week
9
10    // six-argument constructor
11    public BasePlusCommissionEmployee4( String first, String last,
12       String ssn, double sales, double rate, double salary )
13    {
14       super( first, last, ssn, sales, rate );
15       setBaseSalary( salary ); // validate and store base salary
16    } // end six-argument BasePlusCommissionEmployee4 constructor
17
18    // set base salary
19    public void setBaseSalary( double salary )
20    {
21       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
22    } // end method setBaseSalary
23
```

Inherits from CommissionEmployee3

Outline

```
24    // return base salary
25    public double getBaseSalary()
26    {
27       return baseSalary;
28    } // end method getBaseSalary
29
30    // calculate earnings
31    public double earnings()
32    {
33       return getBaseSalary() + super.earnings();
34    } // end method earnings
35
36    // return String representation of BasePlusCommissionEmploye
37    public String toString()
38    {
39       return String.format( "%s %s\n%s: %.2f", "base-salaried",
40          super.toString(), "base salary", getBaseSalary() );
41    } // end method toString
42 } // end class BasePlusCommissionEmployee4
```

Invoke an overridden superclass method from a subclass

(2 of 2)

Use *get* methods to obtain the values of instance variables

Lines 40

Invoke an overridden superclass method from a subclass

Common Programming Error 9.3

When a superclass method is overridden in a subclass, the subclass version often calls the superclass version to do a portion of the work. Failure to prefix the superclass method name with the keyword `super` and a dot (.) separator when referencing the superclass's method causes the subclass method to call itself, creating an error called infinite recursion. Recursion, used correctly, is a powerful capability discussed in Chapter 15, Recursion.

Outline

```
1  // Fig. 9.14: BasePlusCommissionEmployeeTest4.java
2  // Testing class BasePlusCommissionEmployee4.
3
4  public class BasePlusCommissionEmployeeTest4
5  {
6     public static void main( String args[] )
7     {
8        // instantiate BasePlusCommissionEmployee4 object
9        BasePlusCommissionEmployee4 employee =
10          new BasePlusCommissionEmployee4(
11          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
12
13       // get base-salaried commission employee data
14       System.out.println(
15          "Employee information obtained by get methods: \n" );
16       System.out.printf( "%s %s\n", "First name is",
17          employee.getFirstName() );
18       System.out.printf( "%s %s\n", "Last name is",
19          employee.getLastName() );
20       System.out.printf( "%s %s\n", "Social security number is",
21          employee.getSocialSecurityNumber() );
22       System.out.printf( "%s %.2f\n", "Gross sales is",
23          employee.getGrossSales() );
24       System.out.printf( "%s %.2f\n", "Commission rate is",
25          employee.getCommissionRate() );
26       System.out.printf( "%s %.2f\n", "Base salary
27          employee.getBaseSalary() );
28
```

Create `BasePlusCommissionEmployee4` object.

Lines 9-11

Lines 16-25

Use inherited *get* methods to access inherited `private` instance variables

Use `BasePlusCommissionEmployee4` *get* method to access `private` instance variable.

Outline

```
29       employee.setBaseSalary( 1000 ); // set base salary
30
31       System.out.printf( "\n%s:\n\n%s\n",
32          "Updated employee information obtaine
33          employee.toString() );
34    } // end main
35 } // end class BasePlusCommissionEmployeeTest4
```

Use `BasePlusCommissionEmployee4` *set* method to modify `private` instance variable `baseSalary`.

EmployeeTest4.java

(2 of 2)

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Constructors in Subclasses

- ***Example will be skipped***

- **Instantiating subclass object**
  - **Chain of constructor calls**
    - **subclass constructor invokes superclass constructor**
      - **Implicitly or explicitly**
    - **Base of inheritance hierarchy**
      - **Last constructor called in chain is `Object`'s constructor**
      - **Original subclass constructor's body finishes executing last**
      - **Example: `CommissionEmployee3-BasePlusCommissionEmployee4` hierarchy**
        - **`CommissionEmployee3` constructor called second last (last is `Object` constructor)**
        - **`CommissionEmployee3` constructor's body finishes execution second (first is `Object` constructor's body)**

Outline

CommissionEmployee4.java

(1 of 4)

Lines 23-24

```
1  // Fig. 9.15: CommissionEmployee4.java
2  // CommissionEmployee4 class represents a commission employee.
3
4  public class CommissionEmployee4
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee4( String first, String last, String ssn,
14       double sales, double rate )
15    {
16       // implicit call to Object constructor occurs here
17       firstName = first;
18       lastName = last;
19       socialSecurityNumber = ssn;
20       setGrossSales( sales ); // validate ar
21       setCommissionRate( rate ); // validate
22
23       System.out.printf(
24          "\nCommissionEmployee4 constructor:\n%s\n", this );
25    } // end five-argument CommissionEmployee4 constructor
26
```

Constructor outputs message to demonstrate method call order.

Outline

CommissionEmployee4.java

(2 of 4)

```
27    // set first name
28    public void setFirstName( String first )
29    {
30       firstName = first;
31    } // end method setFirstName
32
33    // return first name
34    public String getFirstName()
35    {
36       return firstName;
37    } // end method getFirstName
38
39    // set last name
40    public void setLastName( String last )
41    {
42       lastName = last;
43    } // end method setLastName
44
45    // return last name
46    public String getLastName()
47    {
48       return lastName;
49    } // end method getLastName
50
51    // set social security number
52    public void setSocialSecurityNumber( String ssn )
53    {
54       socialSecurityNumber = ssn; // should validate
55    } // end method setSocialSecurityNumber
56
```

Outline

CommissionEmployee4.java

(3 of 4)

```
57    // return social security number
58    public String getSocialSecurityNumber()
59    {
60       return socialSecurityNumber;
61    } // end method getSocialSecurityNumber
62
63    // set gross sales amount
64    public void setGrossSales( double sales )
65    {
66       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
67    } // end method setGrossSales
68
69    // return gross sales amount
70    public double getGrossSales()
71    {
72       return grossSales;
73    } // end method getGrossSales
74
75    // set commission rate
76    public void setCommissionRate( double rate )
77    {
78       commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
79    } // end method setCommissionRate
80
```

Outline

CommissionEmployee4.java

(4 of 4)

```
81    // return commission rate
82    public double getCommissionRate()
83    {
84       return commissionRate;
85    } // end method getCommissionRate
86
87    // calculate earnings
88    public double earnings()
89    {
90       return getCommissionRate() * getGrossSales();
91    } // end method earnings
92
93    // return String representation of CommissionEmployee4 object
94    public String toString()
95    {
96       return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
97          "commission employee", getFirstName(), getLastName(),
98          "social security number", getSocialSecurityNumber(),
99          "gross sales", getGrossSales(),
100         "commission rate", getCommissionRate() );
101    } // end method toString
102 } // end class CommissionEmployee4
```

Outline

BasePlusCommissionEmployee5.java

(1 of 2)

Lines 15-16

```
1  // Fig. 9.16: BasePlusCommissionEmployee5.java
2  // BasePlusCommissionEmployee5 class declaration.
3
4  public class BasePlusCommissionEmployee5 extends CommissionEmployee4
5  {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee5( String first, String last,
10       String ssn, double sales, double rate, double salary )
11    {
12       super( first, last, ssn, sales, rate )
13       setBaseSalary( salary ); // validate a
14
15       System.out.printf(
16          "\nBasePlusCommissionEmployee5 constructor:\n%s\n", this );
17    } // end six-argument BasePlusCommissionEmployee5 constructor
18
19    // set base salary
20    public void setBaseSalary( double salary )
21    {
22       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23    } // end method setBaseSalary
24
```

Constructor outputs message to demonstrate method call order.

Outline

BasePlusCommissionEmployee5.java

(2 of 2)

```
25    // return base salary
26    public double getBaseSalary()
27    {
28       return baseSalary;
29    } // end method getBaseSalary
30
31    // calculate earnings
32    public double earnings()
33    {
34       return getBaseSalary() + super.earnings();
35    } // end method earnings
36
37    // return String representation of BasePlusCommissionEmployee5
38    public String toString()
39    {
40       return String.format( "%s %s\n%s: %.2f", "base-salaried",
41          super.toString(), "base salary", getBaseSalary() );
42    } // end method toString
43 } // end class BasePlusCommissionEmployee5
```

Outline

```
1  // Fig. 9.17: ConstructorTest.java
2  // Display order in which superclass and subclass constructors are called.
3
4  public class ConstructorTest
5  {
6     public static void main( String args[] )
7     {
8        CommissionEmployee4 employee1 = new CommissionEmployee4(
9           "Bob", "Lewis", "333-33-3333", 5000, .04 );
10
11       System.out.println();
12       BasePlusCommissionEmployee5 employee2 =
13          new BasePlusCommissionEmployee5(
14          "Lisa", "Jones", "555-55-5555", 2000, .06, 800 );
15
16       System.out.println();
17       BasePlusCommissionEmployee5 employee3 =
18          new BasePlusCommissionEmployee5(
19          "Mark", "Sands", "888-88-8888", 8000, .15, 2000 );
20    } // end main
21 } // end class ConstructorTest
```

Instantiate `CommissionEmployee4` object

Instantiate two `BasePlusCommissionEmployee5` objects to demonstrate order of subclass and superclass constructor method calls.

`ctorTest`

`.java`

(1 of 2)

```
CommissionEmployee4 constructor:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

CommissionEmployee4 constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 0.00

BasePlusCommissionEmployee5 constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00


CommissionEmployee4 constructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 0.00

BasePlusCommissionEmployee5 constructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00
```

Subclass `BasePlusCommissionEmployee5` constructor body executes after superclass `CommissionEmployee4`'s constructor finishes execution.

Outline

`ConstructorTest.java`

(2 of 2)

Software Engineering with Inheritance

- **Customizing existing software**
  - **Inherit from existing classes**
    - **Include additional members**
    - **Redefine superclass members**
    - **No direct access to superclass's source code**
      - **Link to object code**
  - **Independent software vendors (ISVs)**
    - **Develop proprietary code for sale/license**
      - **Available in object-code format**
    - **Users derive new classes**
      - **Without accessing ISV proprietary source code**

Object Class

- Class Object methods
  - clone
  - equals
  - finalize
  - getClass
  - hashCode
  - notify, notifyAll, wait
  - toString

Method	Description
`clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it is called. When cloning is required for objects of a class, the class should override method `clone` as a `public` method and should implement interface `Cloneable` (package `java.lang`). The default implementation of this method performs a so-called shallow copy—instance variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a deep copy that creates a new object for each reference type instance variable. There are many subtleties to overriding method `clone`. You can learn more about cloning in the following article: `java.sun.com/developer/JDCTechTips/2001/tt0306.html`

Fig. 9.18 | `Object` methods that are inherited directly or indirectly by all classes. (Part 1 of 4)

Method	Description
`Equals`	This method compares two objects for equality and returns `true` if they are equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the contents of the two objects. The method's implementation should meet the following requirements: • It should return `false` if the argument is `null`. • It should return `true` if an object is compared to itself, as in `object1.equals( object1 )`. • It should return `true` only if both `object1.equals( object2 )` and `object2.equals( object1 )` would return `true`. • For three objects, if `object1.equals( object2 )` returns `true` and `object2.equals( object3 )` returns `true`, then `object1.equals( object3 )` should also return `true`. • If `equals` is called multiple times with the two objects and the objects do not change, the method should consistently return `true` if the objects are equal and `false` otherwise. A class that overrides `equals` should also override `hashCode` to ensure that equal objects have identical hashcodes. The default `equals` implementation uses operator == to determine whether two references *refer to the same object* in memory. Section 29.3.3 demonstrates class `String`'s `equals` method and differentiates between comparing `String` objects with == and with `equals`.

Fig. 9.18 | `Object` methods that are inherited directly or indirectly by all classes. (Part 2 of 4)

Method	Description
`finalize`	This `protected` method (introduced in Section 8.10 and Section 8.11) is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. It is not guaranteed that the garbage collector will reclaim an object, so it cannot be guaranteed that the object's `finalize` method will execute. The method must specify an empty parameter list and must return `void`. The default implementation of this method serves as a placeholder that does nothing.
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` (used in Section 10.5 and Section 21.3) returns an object of class `Class` (package `java.lang`) that contains information about the object's type, such as its class name (returned by `Class` method `getName`). You can learn more about class `Class` in the online API documentation at `java.sun.com/j2se/5.0/docs/api/java/lang/Class.html`.

Fig. 9.18 | `Object` methods that are inherited directly or indirectly by all classes. (Part 3 of 4)

Method	Description
`hashCode`	A hashtable is a data structure (discussed in Section 19.10) that relates one object, called the key, to another object, called the value. When initially inserting a value into a hashtable, the key's `hashCode` method is called. The hashcode value returned is used by the hashtable to determine the location at which to insert the corresponding value. The key's hashcode is also used by the hashtable to locate the key's corresponding value.
`notify`, `notifyAll`, `wait`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 23. In J2SE 5.0, the multithreading model has changed substantially, but these features continue to be supported.
`toString`	This method (introduced in Section 9.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's `hashCode` method.

Fig. 9.18 | `Object` methods that are inherited directly or indirectly by all classes. (Part 4 of 4)

Object-Oriented Programming: Polymorphism

WE WILL COVER

- The concept of polymorphism.
- To use overridden methods to effect polymorphism.
- To distinguish between abstract and concrete classes.
- To declare abstract methods to create abstract classes.
- How polymorphism makes systems extensible and maintainable.
- To determine an object's type at execution time.
- To declare and implement interfaces.

Introduction

- **Polymorphism**
  - **Enables “programming in the general”**
  - **The same invocation can produce “many forms” of results**
- **Interfaces**
  - **Implemented by classes to assign common functionality to possibly unrelated classes**

Polymorphism Examples

- **Polymorphism**
  - **When a program invokes a method through a superclass variable, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable**
  - **The same method name and signature can cause different actions to occur, depending on the type of object on which the method is invoked**
  - **Facilitates adding new classes to a system with minimal modifications to the system's code**

Software Engineering Observation 10.1

Polymorphism enables programmers to deal in generalities and let the execution-time environment handle the specifics. Programmers can command objects to behave in manners appropriate to those objects, without knowing the types of the objects (as long as the objects belong to the same inheritance hierarchy).

Software Engineering Observation 10.2

Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without requiring modification of the base system. Only client code that instantiates new objects must be modified to accommodate new types.

Demonstrating Polymorphic Behavior

- **A superclass reference can be aimed at a subclass object**
  - **This is possible because a subclass object *is a* superclass object as well**
  - **When invoking a method from that reference, the type of the actual referenced object, not the type of the reference, determines which method is called**
- **A subclass reference can be aimed at a superclass object only if the object is downcasted**

Outline

PolymorphismTest.java

(1 of 2)

```
1  // Fig. 10.1: PolymorphismTest.java
2  // Assigning superclass and subclass references to superclass and
3  // subclass variables.
4
5  public class PolymorphismTest
6  {
7     public static void main( String args[] )
8     {
9        // assign superclass reference to superclass variable
10       CommissionEmployee3 commissionEmployee = new CommissionEmployee3(
11          "Sue", "Jones", "222-22-2222", 10000, .06 );
12
13       // assign subclass reference to subclass variable
14       BasePlusCommissionEmployee4 basePlusCommissionEmployee =
15          new BasePlusCommissionEmployee4(
16          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
17
18       // invoke toString on superclass object using superclass variable
19       System.out.printf( "%s %s:\n\n%s\n\n",
20          "Call CommissionEmployee3's toString with superclass reference ",
21          "to superclass object", commissionEmployee.toString() );
22
23       // invoke toString on subclass object using subclass variable
24       System.out.printf( "%s %s:\n\n%s\n\n",
25          "Call BasePlusCommissionEmployee4's toString with subclass",
26          "reference to subclass object",
27          basePlusCommissionEmployee.toString() );
28
```

Typical reference assignments

```
29       // invoke toString on subclass object using super
30       CommissionEmployee3 commissionEmployee2 =
31          basePlusCommissionEmployee;
32       System.out.printf( "%s %s:\n\n%s\n",
33          "Call BasePlusCommissionEmployee4's toString with superclass",
34          "reference to subclass object", commissionEmployee2.toString() );
35    } // end main
36 } // end class PolymorphismTest
```

Assign a reference to a **basePlusCommissionEmployee** object to a **CommissionEmployee3** variable

PolymorphismTest.java

Polymorphically call **basePlusCommissionEmployee**'s **toString** method

```
Call CommissionEmployee3's toString with superclass reference to superclass
object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee4's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee4's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

2)

Abstract Classes and Methods

- **Abstract classes**
  - **Classes that are too general to create real objects**
  - **Used only as abstract superclasses for concrete subclasses and to declare reference variables**
  - **Many inheritance hierarchies have abstract superclasses occupying the top few levels**
  - **Keyword `abstract`**
    - **Use to declare a class `abstract`**
    - **Also use to declare a method `abstract`**
      - **Abstract classes normally contain one or more abstract methods**
      - **All concrete subclasses must override all inherited abstract methods**

Abstract Classes and Methods (Cont.)

- `Iterator` class
  - Traverses all the objects in a collection, such as an array
  - Often used in polymorphic programming to traverse a collection that contains references to objects from various levels of a hierarchy

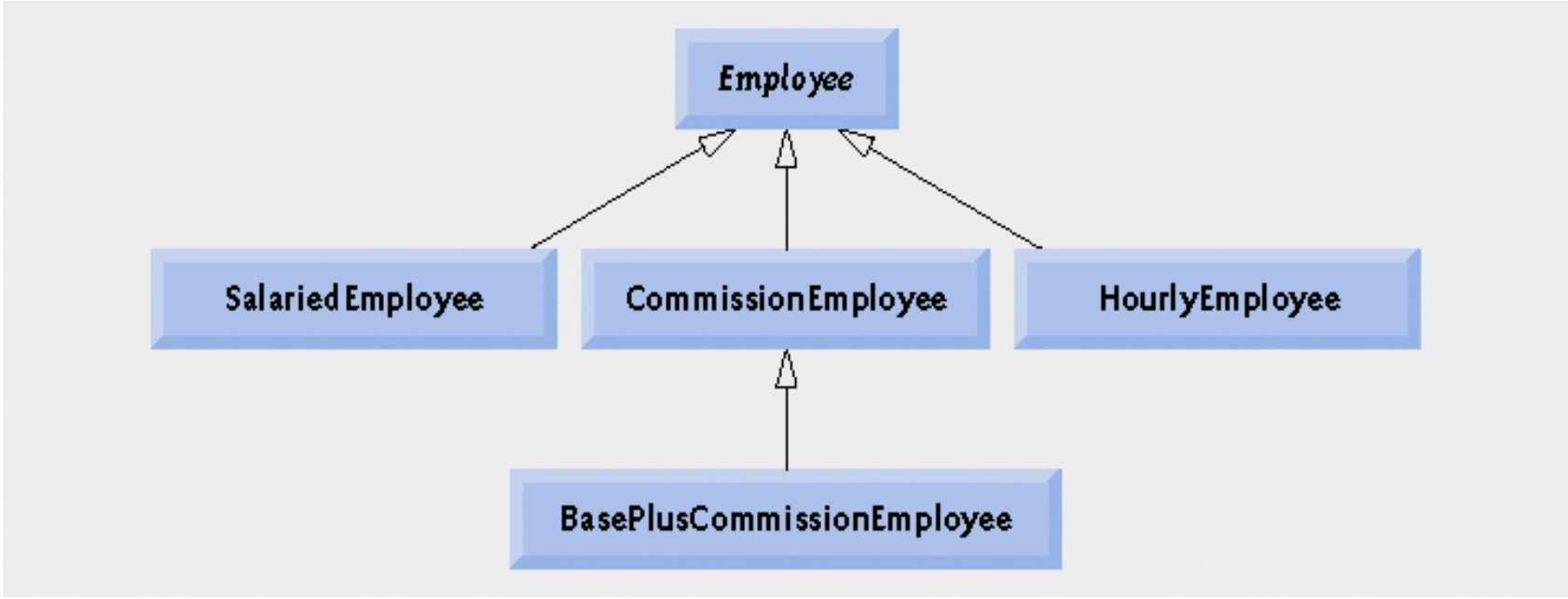


Fig. 10.2 | `Employee` hierarchy UML class diagram.

Software Engineering Observation 10.4

A subclass can inherit "interface" or "implementation" from a superclass. Hierarchies designed for `implementation inheritance` tend to have their functionality high in the hierarchy—each new subclass inherits one or more methods that were implemented in a superclass, and the subclass uses the superclass implementations. (cont…)

Software Engineering Observation 10.4

Hierarchies designed for `interface` `inheritance` tend to have their functionality lower in the hierarchy—a superclass specifies one or more abstract methods that must be declared for each concrete class in the hierarchy, and the individual subclasses override these methods to provide subclass-specific implementations.

Creating Abstract Superclass `Employee`

- **`abstract` superclass `Employee`**
  - **`earnings` is declared `abstract`**
    - **No implementation can be given for `earnings` in the `Employee` `abstract` class**
  - **An array of `Employee` variables will store references to subclass objects**
    - **`earnings` method calls from these variables will call the appropriate version of the `earnings` method**

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* 40 * wage + (hours - 40) * wage * 1.5	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 10.3 | Polymorphic interface for the `Employee` hierarchy classes.

Outline

```
1  // Fig. 10.4: Employee.java
2  // Employee abstract superclass.
3
4  public abstract class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // three-argument constructor
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // end three-argument Employee constructor
17
```

Declare **`abstract`** class **`Employee`**

Attributes common to all employees

`Employee.java`

(1 of 3)

```
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first;
22    } // end method setFirstName
23
24    // return first name
25    public String getFirstName()
26    {
27       return firstName;
28    } // end method getFirstName
29
30    // set last name
31    public void setLastName( String last )
32    {
33       lastName = last;
34    } // end method setLastName
35
36    // return last name
37    public String getLastName()
38    {
39       return lastName;
40    } // end method getLastName
41
```

Outline

Employee.java

(2 of 3)

Outline

Employee.java

(3 of 3)

```
42    // set social security number
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // should validate
46    } // end method setSocialSecurityNumber
47
48    // return social security number
49    public String getSocialSecurityNumber()
50    {
51       return socialSecurityNumber;
52    } // end method getSocialSecurityNumber
53
54    // return String representation of Employee object
55    public String toString()
56    {
57       return String.format( "%s %s\nsocial security number: %s",
58          getFirstName(), getLastName(), getSocialSecurityNumber() );
59    } // end method toString
60
61    // abstract method overridden by subclasses
62    public abstract double earnings(); // no implementation here
63 } // end abstract class Employee
```

`abstract` method **`earnings`** has no implementation

Outline

```
1  // Fig. 10.5: SalariedEmployee.java
2  // SalariedEmployee class extends Employee.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // four-argument constructor
9     public SalariedEmployee( String first, String last, String ssn,
10       double salary )
11    {
12       super( first, last, ssn ); // pass to Employee constructor
13       setWeeklySalary( salary ); // validate and store salary
14    } // end four-argument SalariedEmployee constructor
15
16    // set salary
17    public void setWeeklySalary( double salary )
18    {
19       weeklySalary = salary < 0.0 ? 0.0 : salary;
20    } // end method setWeeklySalary
21
```

Class **SalariedEmployee** extends class **Employee**

SalariedEmployee.java

Call superclass constructor

(1 of 2)

Call **setWeeklySalary** method

Validate and set weekly salary value

Outline

SalariedEmployee.java

(2 of 2)

```
22    // return salary
23    public double getWeeklySalary()
24    {
25       return weeklySalary;
26    } // end method getWeeklySalary
27
28    // calculate earnings; override abstract method earnings in Employee
29    public double earnings()
30    {
31       return getWeeklySalary();
32    } // end method earnings
33
34    // return String representation of SalariedEmployee object
35    public String toString()
36    {
37       return String.format( "salaried employee: %s\n%s: $%,.2f",
38          super.toString(), "weekly salary", getWeeklySalary() );
39    } // end method toString
40 } // end class SalariedEmployee
```

Override **earnings** method so **SalariedEmployee** can be concrete

Override **toString** method

Call superclass's version of **toString**

Outline

HourlyEmployee.java

(1 of 2)

```
1  // Fig. 10.6: HourlyEmployee.java
2  // HourlyEmployee class extends Employee.
3
4  public class HourlyEmployee extends Employee
5  {
6     private double wage; // wage per hour
7     private double hours; // hours worked for week
8
9     // five-argument constructor
10    public HourlyEmployee( String first, String last, String ssn,
11       double hourlyWage, double hoursWorked )
12    {
13       super( first, last, ssn );
14       setWage( hourlyWage ); // validate hourly wage
15       setHours( hoursWorked ); // validate hours worked
16    } // end five-argument HourlyEmployee constructor
17
18    // set wage
19    public void setWage( double hourlyWage )
20    {
21       wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
22    } // end method setWage
23
24    // return wage
25    public double getWage()
26    {
27       return wage;
28    } // end method getWage
29
```

Class **HourlyEmployee** extends class **Employee**

Call superclass constructor

Validate and set hourly wage value

Outline

HourlyEmployee .java

(2 of 2)

```
30    // set hours worked
31    public void setHours( double hoursWorked )
32    {
33       hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
34          hoursWorked : 0.0;
35    } // end method setHours
36
37    // return hours worked
38    public double getHours()
39    {
40       return hours;
41    } // end method getHours
42
43    // calculate earnings; override abstract method earnings in Employee
44    public double earnings()
45    {
46       if ( getHours() <= 40 ) // no overtime
47          return getWage() * getHours();
48       else
49          return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
50    } // end method earnings
51
52    // return String representation of HourlyEmployee object
53    public String toString()
54    {
55       return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
56          super.toString(), "hourly wage", getWage(),
57          "hours worked", getHours() );
58    } // end method toString
59 } // end class HourlyEmployee
```

Validate and set hours worked value

Override **earnings** method so **HourlyEmployee** can be concrete

Override **toString** method

Call superclass's **toString** method

Outline

```
1  // Fig. 10.7: CommissionEmployee.java
2  // CommissionEmployee class extends Employee.
3
4  public class CommissionEmployee extends Employee
5  {
6     private double grossSales; // gross weekly sales
7     private double commissionRate; // commission percentage
8
9     // five-argument constructor
10    public CommissionEmployee( String first, String last, String ssn,
11       double sales, double rate )
12    {
13       super( first, last, ssn );
14       setGrossSales( sales );
15       setCommissionRate( rate );
16    } // end five-argument CommissionEmployee constructor
17
18    // set commission rate
19    public void setCommissionRate( double rate )
20    {
21       commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
22    } // end method setCommissionRate
23
```

Class **CommissionEmployee** extends class **Employee**

CommissionEmployee.java

(1 of 3)

Call superclass constructor

Validate and set commission rate value

Outline

CommissionEmployee.java

(2 of 3)

```
24    // return commission rate
25    public double getCommissionRate()
26    {
27       return commissionRate;
28    } // end method getCommissionRate
29
30    // set gross sales amount
31    public void setGrossSales( double sales )
32    {
33       grossSales = ( sales < 0.0 ) ? 0.0 : sales;
34    } // end method setGrossSales
35
36    // return gross sales amount
37    public double getGrossSales()
38    {
39       return grossSales;
40    } // end method getGrossSales
41
```

Validate and set the gross sales value

Outline

```
42    // calculate earnings; override abstract method earnings in Employee
43    public double earnings()
44    {
45       return getCommissionRate() * getGrossSales();
46    } // end method earnings
47
48    // return String representation of CommissionEmployee object
49    public String toString()
50    {
51       return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
52          "commission employee", super.toString(),
53          "gross sales", getGrossSales(),
54          "commission rate", getCommissionRate() );
55    } // end method toString
56 } // end class CommissionEmployee
```

Override **earnings** method so **CommissionEmployee** can be concrete

CommissionEmployee.java

(3 of 3)

Override **toString** method

Call superclass's **toString** method

Outline

BasePlusCommissionEmployee.java

(1 of 2)

```
1  // Fig. 10.8: BasePlusCommissionEmp
2  // BasePlusCommissionEmployee clas
3
4  public class BasePlusCommissionEmployee extends CommissionEmployee
5  {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee( String first, String last,
10       String ssn, double sales, double rate, double salary )
11    {
12       super( first, last, ssn, sales, rate );
13       setBaseSalary( salary ); // validate and store base salary
14    } // end six-argument BasePlusCommissionEmployee constructor
15
16    // set base salary
17    public void setBaseSalary( double salary )
18    {
19       baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
20    } // end method setBaseSalary
21
```

Class **BasePlusCommissionEmployee** extends class **CommissionEmployee**

Call superclass constructor

Validate and set base salary value

Outline

BasePlusCommissionEmployee.java

```
22    // return base salary
23    public double getBaseSalary()
24    {
25       return baseSalary;
26    } // end method getBaseSalary
27
28    // calculate earnings; override method earnings in CommissionEmployee
29    public double earnings()
30    {
31       return getBaseSalary() + super.earnings();
32    } // end method earnings
33
34    // return String representation of BasePlusCommissionEmployee object
35    public String toString()
36    {
37       return String.format( "%s %s; %s: $%,.2f",
38          "base-salaried", super.toString(),
39          "base salary", getBaseSalary() );
40    } // end method toString
41 } // end class BasePlusCommissionEmployee
```

Override **earnings** method

Call superclass's **earnings** method

Override **toString** method

Call superclass's **toString** method

(2 of 2)

Outline

PayrollSystemTest.java

(1 of 5)

```
1  // Fig. 10.9: PayrollSystemTest.java
2  // Employee hierarchy test program.
3
4  public class PayrollSystemTest
5  {
6     public static void main( String args[] )
7     {
8        // create subclass objects
9        SalariedEmployee salariedEmployee =
10          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
11       HourlyEmployee hourlyEmployee =
12          new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
13       CommissionEmployee commissionEmployee =
14          new CommissionEmployee(
15          "Sue", "Jones", "333-33-3333", 10000, .06 );
16       BasePlusCommissionEmployee basePlusCommissionEmployee =
17          new BasePlusCommissionEmployee(
18          "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
19
20       System.out.println( "Employees processed individually:\n" );
21
```

Outline

PayrollSystemTest.java

(2 of 5)

```
22       System.out.printf( "%s\n%s: $%,.2f\n\n",
23          salariedEmployee, "earned", salariedEmployee.earnings() );
24       System.out.printf( "%s\n%s: $%,.2f\n\n",
25          hourlyEmployee, "earned", hourlyEmployee.earnings() );
26       System.out.printf( "%s\n%s: $%,.2f\n\n",
27          commissionEmployee, "earned", commissionEmployee.earnings() );
28       System.out.printf( "%s\n%s: $%,.2f\n\n",
29          basePlusCommissionEmployee,
30          "earned", basePlusCommissionEmployee.earnings() );
31
32       // create four-element Employee array
33       Employee employees[] = new Employee[ 4 ];
34
35       // initialize array with Employees
36       employees[ 0 ] = salariedEmployee;
37       employees[ 1 ] = hourlyEmployee;
38       employees[ 2 ] = commissionEmployee;
39       employees[ 3 ] = basePlusCommissionEmployee;
40
41       System.out.println( "Employees processed polymorphically:\n" );
42
43       // generically process each element in array employees
44       for ( Employee currentEmployee : employees )
45       {
46          System.out.println( currentEmployee ); // invokes toString
47
```

Assigning subclass objects to supercalss variables

Implicitly and polymorphically call **`toString`**

Outline

```
48          // determine whether element is a BasePlusCommissionEmployee
49          if ( currentEmployee instanceof BasePlusCommissionEmployee )
50          {
51             // downcast Employee reference to
52             // BasePlusCommissionEmployee reference
53             BasePlusCommissionEmployee employee =
54                ( BasePlusCommissionEmployee ) currentEmployee;
55
56             double oldBaseSalary = employee.getBaseSalary();
57             employee.setBaseSalary( 1.10 * oldBaseSalary );
58             System.out.printf(
59                "new base salary with 10%% increase is: $%,.2f\n",
60                employee.getBaseSalary() );
61          } // end if
62
63          System.out.printf(
64             "earned $%,.2f\n\n", currentEmployee.earnings() );
65       } // end for
66
67       // get type name of each object in employees array
68       for ( int j = 0; j < employees.length; j++ )
69          System.out.printf( "Employee %d is a %s\n", j,
70             employees[ j ].getClass().getName() );
71    } // end main
72 } // end class PayrollSystemTest
```

If the **currentEmployee** variable points to a **BasePlusCommissionEmployee** object

Downcast **currentEmployee** to a **BasePlusCommissionEmployee** reference

Give **BasePlusCommissionEmployee**s a 10% base salary bonus

Polymorphically call **earnings** method

Call **getClass** and **getName** methods to display each **Employee** subclass object's class name

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00
```

Outline

PayrollSystemTest.java

(4 of 5)

Outline

```
Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Same results as when the employees were processed individually

Base salary is increased by 10%

Each employee's type is displayed

PayrollSystemTest.java

(5 of 5)

Demonstrating Polymorphic Processing, Operator `instanceof` and Downcasting

- **Dynamic binding**
  - **Also known as late binding**
  - **Calls to overridden methods are resolved at execution time, based on the type of object referenced**
- **`instanceof` operator**
  - **Determines whether an object is an instance of a certain type**

Demonstrating Polymorphic Processing, Operator `instanceof` and Downcasting

- **Downcasting**
  - **Convert a reference to a superclass to a reference to a subclass**
  - **Allowed only if the object has an *is-a* relationship with the subclass**
- **`getClass` method**
  - **Inherited from `Object`**
  - **Returns an object of type `Class`**
- **`getName` method of class `Class`**
  - **Returns the class's name**

Summary of the Allowed Assignments Between Superclass and Subclass Variables

- **Superclass and subclass assignment rules**
  - **Assigning a superclass reference to a superclass variable is straightforward**
  - **Assigning a subclass reference to a subclass variable is straightforward**
  - **Assigning a subclass reference to a superclass variable is safe because of the *is-a* relationship**
    - **Referring to subclass-only members through superclass variables is a compilation error**
  - **Assigning a superclass reference to a subclass variable is a compilation error**
    - **Downcasting can get around this error**

`final` Methods and Classes

- **`final` methods**
  - **Cannot be overridden in a subclass**
  - **`private` and `static` methods are implicitly `final`**
  - **`final` methods are resolved at compile time, this is known as static binding**
    - **Compilers can optimize by inlining the code**
- **`final` classes**
  - **Cannot be extended by a subclass**
  - **All methods in a `final` class are implicitly `final`**

Case Study: Creating and Using Interfaces

- **Interfaces**
  - **Keyword `interface`**
  - **Contains only constants and `abstract` methods**
    - **All fields are implicitly `public`, `static` and `final`**
    - **All methods are implicitly `public abstract` methods**
  - **Classes can `implement` interfaces**
    - **The class must declare each method in the interface using the same signature or the class must be declared `abstract`**
  - **Typically used when disparate classes need to share common methods and constants**
  - **Normally declared in their own files with the same names as the interfaces and with the `.java` file-name extension**

Developing a `Payable` Hierarchy

- **`Payable` interface**
  - **Contains method `getPaymentAmount`**
  - **Is implemented by the `Invoice` and `Employee` classes**
- **UML representation of interfaces**
  - **Interfaces are distinguished from classes by placing the word "interface" in guillemets (« and ») above the interface name**
  - **The relationship between a class and an interface is known as realization**
    - **A class "realizes" the methods of an interface**

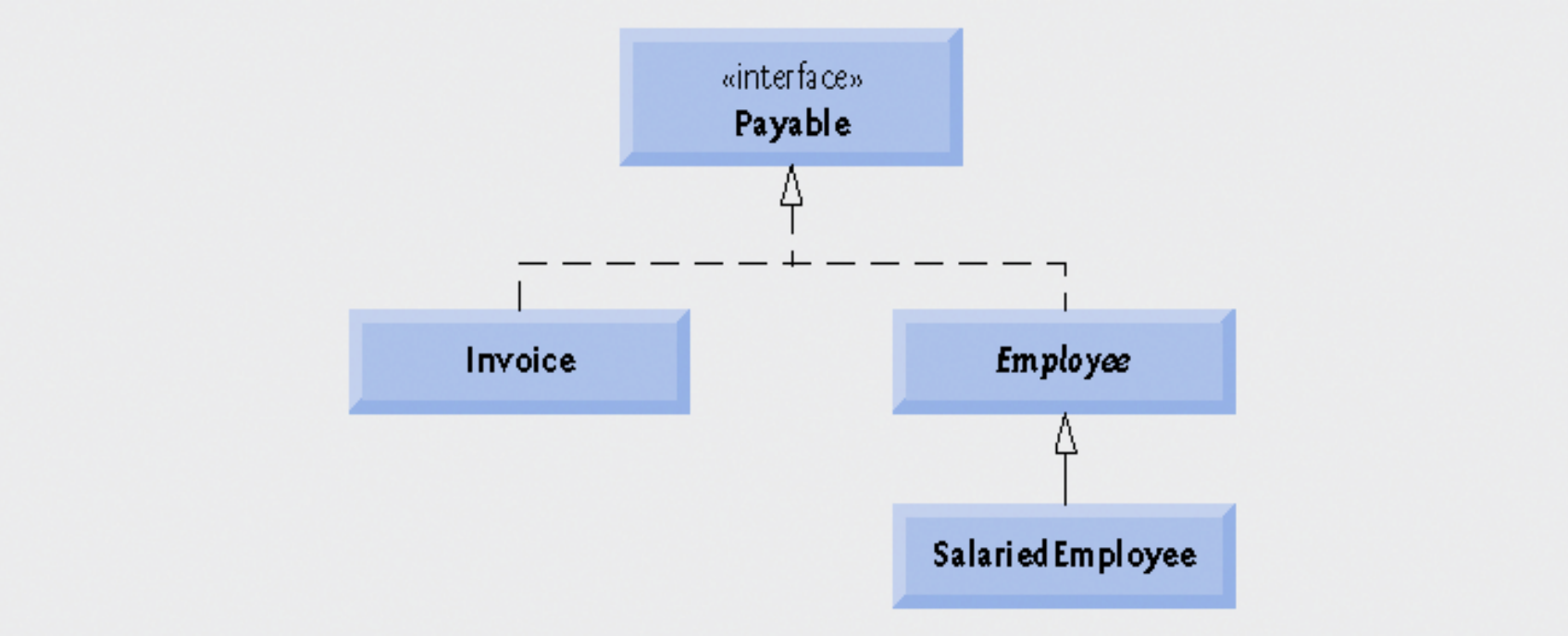


Fig. 10.10 | `Payable` interface hierarchy UML class diagram.

Outline

Payable.java

```
1 // Fig. 10.11: Payable.java
2 // Payable interface declaration.
3
4 public interface Payable
5 {
6    double getPaymentAmount(); // calculate payment; no implementation
7 } // end interface Payable
```

Declare interface **Payable**

Declare **getPaymentAmount** method which is implicitly **public** and **abstract**

Outline

Invoice.java

(1 of 3)

```
1  // Fig. 10.12: Invoice.java
2  // Invoice class implements Payable.
3
4  public class Invoice implements Payable
5  {
6     private String partNumber;
7     private String partDescription;
8     private int quantity;
9     private double pricePerItem;
10
11    // four-argument constructor
12    public Invoice( String part, String description, int count,
13       double price )
14    {
15       partNumber = part;
16       partDescription = description;
17       setQuantity( count ); // validate and store quantity
18       setPricePerItem( price ); // validate and store price per item
19    } // end four-argument Invoice constructor
20
21    // set part number
22    public void setPartNumber( String part )
23    {
24       partNumber = part;
25    } // end method setPartNumber
26
```

Class **Invoice** implements interface **Payable**

Outline

Invoice.java

(2 of 3)

```
27    // get part number
28    public String getPartNumber()
29    {
30       return partNumber;
31    } // end method getPartNumber
32
33    // set description
34    public void setPartDescription( String description )
35    {
36       partDescription = description;
37    } // end method setPartDescription
38
39    // get description
40    public String getPartDescription()
41    {
42       return partDescription;
43    } // end method getPartDescription
44
45    // set quantity
46    public void setQuantity( int count )
47    {
48       quantity = ( count < 0 ) ? 0 : count; // quantity cannot be negative
49    } // end method setQuantity
50
51    // get quantity
52    public int getQuantity()
53    {
54       return quantity;
55    } // end method getQuantity
56
```

Outline

Invoice.java

(3 of 3)

```
57    // set price per item
58    public void setPricePerItem( double price )
59    {
60       pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
61    } // end method setPricePerItem
62
63    // get price per item
64    public double getPricePerItem()
65    {
66       return pricePerItem;
67    } // end method getPricePerItem
68
69    // return String representation of Invoice object
70    public String toString()
71    {
72       return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
73          "invoice", "part number", getPartNumber(), getPartDescription(),
74          "quantity", getQuantity(), "price per item", getPricePerItem() );
75    } // end method toString
76
77    // method required to carry out contract with interface Payable
78    public double getPaymentAmount()
79    {
80       return getQuantity() * getPricePerItem(); // calculate total cost
81    } // end method getPaymentAmount
82 } // end class Invoice
```

Declare **`getPaymentAmount`** to fulfill contract with interface **`Payable`**

Creating Class `Invoice`

- **A class can implement as many interfaces as it needs**
  - **Use a comma-separated list of interface names after keyword implements**
    - **Example: `public class` *ClassName* `extends` *SuperclassName* `implements` *FirstInterface*`,` *SecondInterface*`, ...`**

Outline

Employee.java

(1 of 3)

```
1  // Fig. 10.13: Employee.java
2  // Employee abstract superclass implements Payable.
3
4  public abstract class Employee implements Payable
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // three-argument constructor
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // end three-argument Employee constructor
17
```

Class **Employee** implements interface **Payable**

Outline

Employee.java

(2 of 3)

```
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first;
22    } // end method setFirstName
23
24    // return first name
25    public String getFirstName()
26    {
27       return firstName;
28    } // end method getFirstName
29
30    // set last name
31    public void setLastName( String last )
32    {
33       lastName = last;
34    } // end method setLastName
35
36    // return last name
37    public String getLastName()
38    {
39       return lastName;
40    } // end method getLastName
41
```

Outline

Employee.java

(3 of 3)

```
42    // set social security number
43    public void setSocialSecurityNumber( String ssn )
44    {
45       socialSecurityNumber = ssn; // should validate
46    } // end method setSocialSecurityNumber
47
48    // return social security number
49    public String getSocialSecurityNumber()
50    {
51       return socialSecurityNumber;
52    } // end method getSocialSecurityNumber
53
54    // return String representation of Employee object
55    public String toString()
56    {
57       return String.format( "%s %s\nsocial security number: %s",
58          getFirstName(), getLastName(), getSocialSecurityNumber() );
59    } // end method toString
60
61    // Note: We do not implement Payable method getPaymentAmount here so
62    // this class must be declared abstract to avoid a compilation error.
63 } // end abstract class Employee
```

`getPaymentAmount` method is not implemented here

Modifying Class `SalariedEmployee` for Use in the `Payable` Hierarchy

- **Objects of any subclasses of the class that implements the interface can also be thought of as objects of the interface**
  - **A reference to a subclass object can be assigned to an interface variable if the superclass implements that interface**

Software Engineering Observation 10.7

Inheritance and interfaces are similar in their implementation of the "is-a" relationship. An object of a class that implements an interface may be thought of as an object of that interface type. An object of any subclasses of a class that implements an interface also can be thought of as an object of the interface type.

Outline

```
1  // Fig. 10.14: SalariedEmployee.java
2  // SalariedEmployee class extends Employee, which implements Payable.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // four-argument constructor
9     public SalariedEmployee( String first, String last, String ssn,
10       double salary )
11    {
12       super( first, last, ssn ); // pass to Employee constructor
13       setWeeklySalary( salary ); // validate and store salary
14    } // end four-argument SalariedEmployee constructor
15
16    // set salary
17    public void setWeeklySalary( double salary )
18    {
19       weeklySalary = salary < 0.0 ? 0.0 : salary;
20    } // end method setWeeklySalary
21
```

Class **SalariedEmployee** extends class **Employee** (which implements interface **Payable**)

SalariedEmployee.java

(1 of 2)

Outline

SalariedEmployee.java

```
22    // return salary
23    public double getWeeklySalary()
24    {
25       return weeklySalary;
26    } // end method getWeeklySalary
27
28    // calculate earnings; implement interface Payable method that was
29    // abstract in superclass Employee
30    public double getPaymentAmount()
31    {
32       return getWeeklySalary();
33    } // end method getPaymentAmount
34
35    // return String representation of SalariedEmployee object
36    public String toString()
37    {
38       return String.format( "salaried employee: %s\n%s: $%,.2f",
39          super.toString(), "weekly salary", getWeeklySalary() );
40    } // end method toString
41 } // end class SalariedEmployee
```

Declare **`getPaymentAmount`** method instead of **`earnings`** method

(2 of 2)

Outline

PayableInterface Test.java

```
1  // Fig. 10.15: PayableInterfaceTest.java
2  // Tests interface Payable.
3
4  public class PayableInterfaceTest
5  {
6     public static void main( String args[] )
7     {
8        // create four-element Payable array
9        Payable payableObjects[] = new Payable[ 4 ];
10
11       // populate array with objects that implement Payable
12       payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
13       payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
14       payableObjects[ 2 ] =
15          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
16       payableObjects[ 3 ] =
17          new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );
18
19       System.out.println(
20          "Invoices and Employees processed polymorphically:\n" );
21
```

Declare array of **Payable** variables

Assigning references to **Invoice** objects to **Payable** variables

Assigning references to **SalariedEmployee** objects to **Payable** variables

Outline

PayableInterface Test.java

(2 of 2)

```
22       // generically process each element in array payableObjects
23       for ( Payable currentPayable : payableObjects )
24       {
25          // output currentPayable and its appropriate payment amount
26          System.out.printf( "%s \n%s: $%,.2f\n\n",
27             currentPayable.toString(),
28             "payment due", currentPayable.getPaymentAmount() );
29       } // end for
30    } // end main
31 } // end class PayableInterfaceTest
```

Call **toString** and **getPaymentAmount** methods polymorphically

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Software Engineering Observation 10.10

All methods of class `Object` can be called by using a reference of an interface type. A reference refers to an object, and all objects inherit the methods of class `Object`.

Declaring Constants with Interfaces

- **Interfaces can be used to declare constants used in many class declarations**
  - **These constants are implicitly `public`, `static` and `final`**
  - **Using a `static import` declaration allows clients to use these constants with just their names**

Software Engineering Observation 10.11

It is considered a better programming practice to create sets of constants as enumerations with keyword `enum`. See Section 6.10 for an introduction to `enum` and Section 8.9 for additional `enum` details.

Interface	Description
`Comparable`	As you learned in Chapter 2, Java contains several comparison operators (e.g., <, <=, >, >=, ==, !=) that allow you to compare primitive values. However, these operators cannot be used to compare the contents of objects. Interface `Comparable` is used to allow objects of a class that implements the interface to be compared to one another. The interface contains one method, `compareTo`, that compares the object that calls the method to the object passed as an argument to the method. Classes must implement `compareTo` such that it returns a value indicating whether the object on which it is invoked is less than (negative integer return value), equal to (`0` return value) or greater than (positive integer return value) the object passed as an argument, using any criteria specified by the programmer. For example, if class `Employee` implements `Comparable`, its `compareTo` method could compare `Employee` objects by their earnings amounts. Interface `Comparable` is commonly used for ordering objects in a collection such as an array. We use `Comparable` in Chapter 18, Generics, and Chapter 19, Collections.
`Serializable`	A tagging interface used only to identify classes whose objects can be written to (i.e., serialized) or read from (i.e., deserialized) some type of storage (e.g., file on disk, database field) or transmitted across a network. We use `Serializable` in Chapter 14, Files and Streams, and Chapter 24, Networking.

Fig. 10.16 | Common interfaces of the Java API. (Part 1 of 2)

Interface	Description
`Runnable`	Implemented by any class for which objects of that class should be able to execute in parallel using a technique called multithreading (discussed in Chapter 23, Multithreading). The interface contains one method, `run`, which describes the behavior of an object when executed.
GUI event-listener interfaces	You work with Graphical User Interfaces (GUIs) every day. For example, in your Web browser, you might type in a text field the address of a Web site to visit, or you might click a button to return to the previous site you visited. When you type a Web site address or click a button in the Web browser, the browser must respond to your interaction and perform the desired task for you. Your interaction is known as an event, and the code that the browser uses to respond to an event is known as an event handler. In Chapter 11, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2, you will learn how to build Java GUIs and how to build event handlers to respond to user interactions. The event handlers are declared in classes that implement an appropriate event-listener interface. Each event listener interface specifies one or more methods that must be implemented to respond to user interactions.
`SwingConstants`	Contains a set of constants used in GUI programming to position GUI elements on the screen. We explore GUI programming in Chapters 11 and 22.

Fig. 10.16 | Common interfaces of the Java API. (Part 2 of 2)