10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting (Cont.)

- All calls to method `toString` and `earnings` are resolved at execution time, based on the type of the object to which `currentEmployee` refers.
  - Known as dynamic binding or late binding.
  - Java decides which class's `toString` method to call at execution time rather than at compile time
- A superclass reference can be used to invoke only methods of the superclass—the subclass method implementations are invoked polymorphically.
- Attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

Common Programming Error 10.3

Assigning a superclass variable to a subclass variable (without an explicit cast) is a compilation error.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 10.4

If a subclass object's reference has been assigned to a variable of one of its direct or indirect superclasses at execution time, it's acceptable to downcast the reference stored in that superclass variable back to a subclass-type reference. Before performing such a cast, use the `instanceof` *operator to ensure that the object is indeed an object of an appropriate subclass.*

Common Programming Error 10.4

When downcasting a reference, a `ClassCastException` *occurs if the referenced object at execution time does not have an* is-a *relationship with the type specified in the cast operator.*

10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting (Cont.)

- Every object in Java knows its own class and can access this information through the getClass method, which all classes inherit from class `Object`.
  - The `getClass` method returns an object of type Class (from package `java.lang`), which contains information about the object's type, including its class name.
  - The result of the `getClass` call is used to invoke getName to get the object's class name.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 10.5

Although the actual method that's called depends on the runtime type of the object to which a variable refers, a variable can be used to invoke only those methods that are members of that variable's type, which the compiler verifies.

10.5.7 Summary of the Allowed Assignments Between Superclass and Subclass Variables

- There are four ways to assign superclass and subclass references to variables of superclass and subclass types.
- Assigning a superclass reference to a superclass variable is straightforward.
- Assigning a subclass reference to a subclass variable is straightfor-ward.
- Assigning a subclass reference to a superclass variable is safe, because the subclass object *is an object of its superclass*.
  - The superclass variable can be used to refer only to superclass members.
  - If this code refers to subclass-only mem-bers through the superclass variable, the compiler reports errors.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.5.7 Summary of the Allowed Assignments Between Superclass and Subclass Variables (Cont.)

- Attempting to assign a superclass reference to a subclassvariable is a compilation error.
  - To avoid this error, the superclass reference must be cast to a subclass type explicitly.
  - At *execution time, if the object to which the reference refers is not a subclass object, an exception will occur.*
  - Use the `instanceof` operator to ensure that such a cast is performed only if the object is a subclass object.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.6 `final` Methods and Classes

- A final method in a superclass cannot be overridden in a subclass.
  - Methods that are declared `private` are implicitly `final`, because it's not possible to override them in a subclass.
  - Methods that are declared `static` are implicitly `final`.
  - A `final` method's declaration can never change, so all subclasses use the same method implementation, and calls to `final` methods are resolved at compile time—this is known as static binding.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.6 `final` Methods and Classes (Cont.)

- A final class cannot be a superclass (i.e., a class cannot extend a `final` class).
  - All methods in a `final` class are implicitly `final`.
- Class `String` is an example of a `final` class.
  - If you were allowed to create a subclass of `String`, objects of that subclass could be used wherever `String`s are expected.
  - Since class `String` cannot be extended, programs that use `String`s can rely on the functionality of `String` objects as specified in the Java API.
  - Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

Common Programming Error 10.5

Attempting to declare a subclass of a `final` *class is a compilation error.*

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 10.6

In the Java API, the vast majority of classes are not *declared* `final`. *This enables inheritance and polymorphism. However, in some cases, it's important to declare classes* `final`*—typically for security reasons.*

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.7 Case Study: Creating and Using Interfaces

- Our next example reexamines the payroll system of Section 10.5.
- Suppose that the company involved wishes to perform several accounting operations in a single accounts payable application
  - Calculating the earnings that must be paid to each employee
  - Calculate the payment due on each of several invoices (i.e., bills for goods purchased)
- Both operations have to do with obtaining some kind of payment amount.
  - For an employee, the payment refers to the employee's earnings.
  - For an invoice, the payment refers to the total cost of the goods listed on the invoice.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.7 Case Study: Creating and Using Interfaces (Cont.)

- Interfaces offer a capability requiring that unrelated classes implement a set of common methods.
- Interfaces define and standardize the ways in which things such as people and systems can interact with one another.
  - Example: The controls on a radio serve as an interface between radio users and a radio's internal components.
  - Can perform only a limited set of operations (e.g., change the station, adjust the volume, choose between AM and FM)
  - Different radios may implement the controls in different ways (e.g., using push buttons, dials, voice commands).

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.7 Case Study: Creating and Using Interfaces (Cont.)

- The interface specifies *what* operations a radio must permit users to perform but does not specify *how* the operations are performed.
- A Java interface describes a set of methods that can be called on an object.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.7 Case Study: Creating and Using Interfaces (Cont.)

- An interface declaration begins with the keyword `interface` and contains only constants and `abstract` methods.
  - All interface members must be `public`.
  - Interfaces may not specify any implementation details, such as concrete method declarations and instance variables.
  - All methods declared in an interface are implicitly `public abstract` methods.
  - All fields are implicitly `public`, `static` and `final`.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

Good Programming Practice 10.1

According to Chapter 9 of the Java Language Specification, *it's proper style to declare an interface's methods without keywords* `public` *and* `abstract`, *because they're redundant in interface method declarations. Similarly, constants should be declared without keywords* `public`, `static` *and* `final`, *because they, too, are redundant.*

10.7 Case Study: Creating and Using Interfaces (Cont.)

- To use an interface, a concrete class must specify that it implements the interface and must declare each method in the interface with specified signature.
  - Add the `implements` keyword and the name of the interface to the end of your class declaration's first line.
- A class that does not implement all the methods of the interface is an abstract class and must be declared `abstract`.
- Implementing an interface is like signing a contract with the compiler that states, "I will declare all the methods specified by the interface or I will declare my class `abstract`."

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

Common Programming Error 10.6

Failing to implement any method of an interface in a concrete class that `implements` *the interface results in a compilation error indicating that the class must be declared* `abstract`.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.7 Case Study: Creating and Using Interfaces (Cont.)

- An interface is often used when disparate (i.e., unrelated) classes need to share common methods and constants.
  - Allows objects of unrelated classes to be processed polymorphically by responding to the same method calls.
  - You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.7 Case Study: Creating and Using Interfaces (Cont.)

- An interface is often used in place of an `abstract` class when there is no default implementation to inherit—that is, no fields and no default method implementations.
- Like `public abstract` classes, interfaces are typically `public` types.
- A `public` interface must be declared in a file with the same name as the interface and the `.java` file-name extension.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.7.1 Developing a `Payable` Hierarchy

- Next example builds an application that can determine payments for employees and invoices alike.
  - Classes `Invoice` and `Employee` both represent things for which the company must be able to calculate a payment amount.
  - Both classes implement the `Payable` interface, so a program can invoke method `getPaymentAmount` on `Invoice` objects and `Employee` objects alike.
  - Enables the polymorphic processing of `Invoice`s and `Employee`s.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

Good Programming Practice 10.2

When declaring a method in an interface, choose a method name that describes the method's purpose in a general *manner, because the method may be implemented by many unrelated classes.*

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.7.1 Developing a Payable Hierarchy (Cont.)

- Fig. 10.10 shows the accounts payable hierarchy.
- The UML distinguishes an interface from other classes by placing «interface» above the interface name.
- The UML expresses the relationship between a class and an interface through a realization.
  - A class is said to "realize," or implement, the methods of an interface.
  - A class diagram models a realization as a dashed arrow with a hollow arrowhead pointing from the implementing class to the interface.
- A subclass inherits its superclass's realization relationships.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

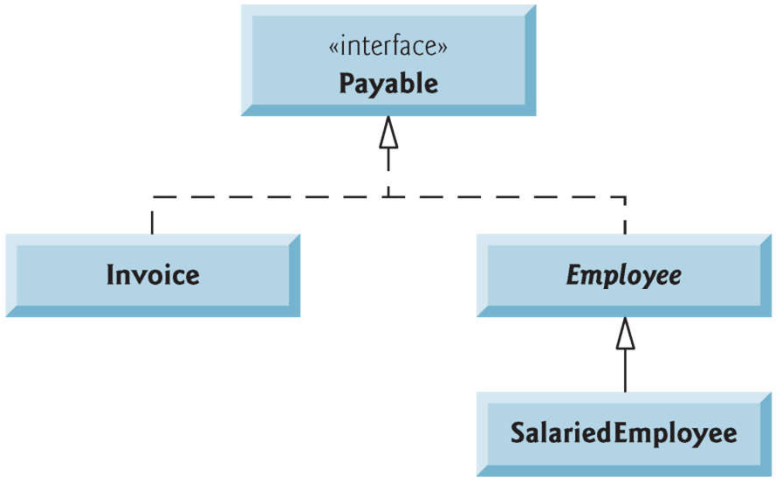


Fig. 10.10 | `Payable` interface hierarchy UML class diagram.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.7.2 Interface `Payable`

- Fig. 10.11 shows the declaration of interface `Payable`.
- Interface methods are always `public` and `abstract`, so they do not need to be declared as such.
- Interfaces can have any number of methods.
- Interfaces may also contain fields that are implicitly `final` and `static`.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```
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
```

Fig. 10.11 | `Payable` interface declaration.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.7.3 Class `Invoice`

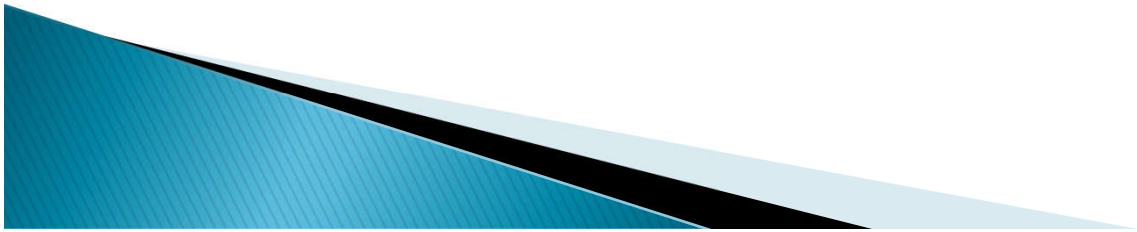
- Java does not allow subclasses to inherit from more than one superclass, but it allows a class to inherit from one superclass and implement as many interfaces as it needs.
- To implement more than one interface, use a comma-separated list of interface names after keyword `implements` in the class declaration, as in:

```
public class ClassName extends SuperclassName
   implements FirstInterface, SecondInterface, ...
```

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 10.12: Invoice.java
2  // Invoice class that 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
```

Fig. 10.12 | `Invoice` class that implements `Payable`. (Part 1 of 4.)



```
21      // set part number
22      public void setPartNumber( String part )
23      {
24         partNumber = part; // should validate
25      } // end method setPartNumber
26
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; // should validate
37      } // end method setPartDescription
38
39      // get description
40      public String getPartDescription()
41      {
42         return partDescription;
43      } // end method getPartDescription
44
```

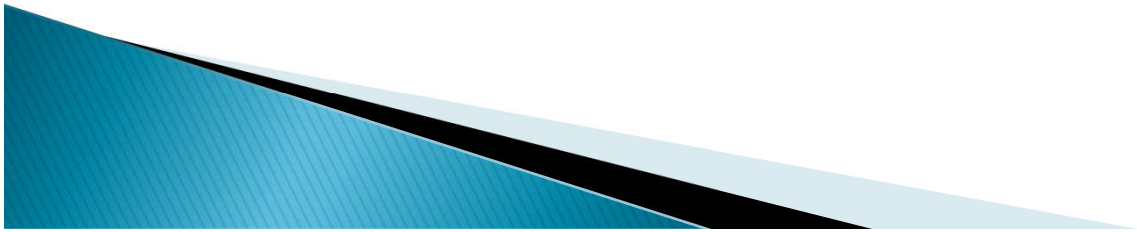
Fig. 10.12 | `Invoice` class that implements `Payable`. (Part 2 of 4.)

```
45      // set quantity
46      public void setQuantity( int count )
47      {
48         if ( count >= 0 )
49            quantity = count;
50         else
51            throw new IllegalArgumentException( "Quantity must be >= 0" );
52      } // end method setQuantity
53
54      // get quantity
55      public int getQuantity()
56      {
57         return quantity;
58      } // end method getQuantity
59
60      // set price per item
61      public void setPricePerItem( double price )
62      {
63         if ( price >= 0.0 )
64            pricePerItem = price;
65         else
66            throw new IllegalArgumentException(
67               "Price per item must be >= 0" );
68      } // end method setPricePerItem
```

Fig. 10.12 | `Invoice` class that implements `Payable`. (Part 3 of 4.)

```
69
70      // get price per item
71      public double getPricePerItem()
72      {
73         return pricePerItem;
74      } // end method getPricePerItem
75
76      // return String representation of Invoice object
77      @Override
78      public String toString()
79      {
80         return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
81            "invoice", "part number", getPartNumber(), getPartDescription(),
82            "quantity", getQuantity(), "price per item", getPricePerItem() );
83      } // end method toString
84
85      // method required to carry out contract with interface Payable
86      @Override
87      public double getPaymentAmount()
88      {
89         return getQuantity() * getPricePerItem(); // calculate total cost
90      } // end method getPaymentAmount
91   } // end class Invoice
```

Fig. 10.12 | `Invoice` class that implements `Payable`. (Part 4 of 4.)



Software Engineering Observation 10.7

All objects of a class that implement multiple interfaces have the is-a *relationship with each implemented interface type.*

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

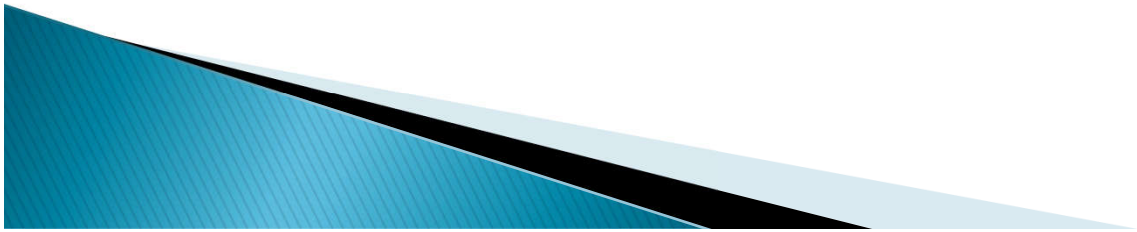
10.7.4 Modifying Class `Employee` to Implement Interface `Payable`

- When a class implements an interface, it makes a contract with the compiler
  - The class will implement each of the methods in the interface or that the class will be declared `abstract`.
  - If the latter, we do not need to declare the interface methods as `abstract` in the `abstract` class—they are already implicitly declared as such in the interface.
  - Any concrete subclass of the `abstract` class must implement the interface methods to fulfill the contract.
  - If the subclass does not do so, it too must be declared `abstract`.
- Each direct `Employee` subclass inherits the superclass's contract to implement method `getPaymentAmount` and thus must implement this method to become a concrete class for which objects can be instantiated.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```
1  // Fig. 10.13: Employee.java
2  // Employee abstract superclass that 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
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first; // should validate
22    } // end method setFirstName
23
```

Fig. 10.13 | `Employee` class that implements `Payable`. (Part 1 of 3.)

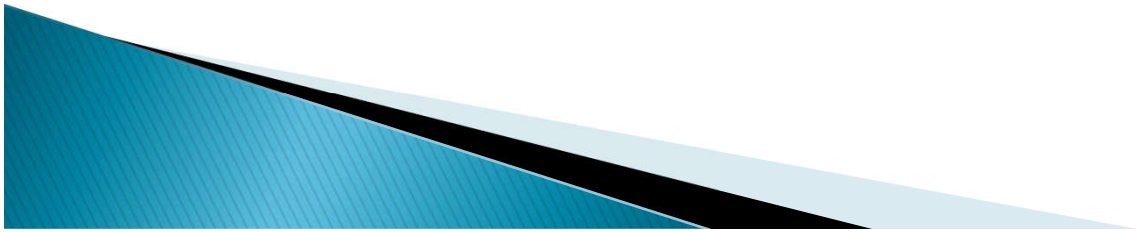


```
24      // return first name
25      public String getFirstName()
26      {
27         return firstName;
28      } // end method getFirstName
29
30      // set last name
31      public void setLastName( String last )
32      {
33         lastName = last; // should validate
34      } // end method setLastName
35
36      // return last name
37      public String getLastName()
38      {
39         return lastName;
40      } // end method getLastName
41
42      // set social security number
43      public void setSocialSecurityNumber( String ssn )
44      {
45         socialSecurityNumber = ssn; // should validate
46      } // end method setSocialSecurityNumber
47
```

Fig. 10.13 | `Employee` class that implements `Payable`. (Part 2 of 3.)

```
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      @Override
56      public String toString()
57      {
58         return String.format( "%s %s\nsocial security number: %s",
59            getFirstName(), getLastName(), getSocialSecurityNumber() );
60      } // end method toString
61
62      // Note: We do not implement Payable method getPaymentAmount here so
63      // this class must be declared abstract to avoid a compilation error.
64   } // end abstract class Employee
```

Fig. 10.13 | `Employee` class that implements `Payable`. (Part 3 of 3.)



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

- Figure 10.14 contains a modified `SalariedEmployee` class that extends `Employee` and fulfills superclass `Employee`'s contract to implement `Payable` method `getPayment-Amount`.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

Software Engineering Observation 10.8

When a method parameter is declared with a superclass or interface type, the method processes the object received as an argument polymorphically.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

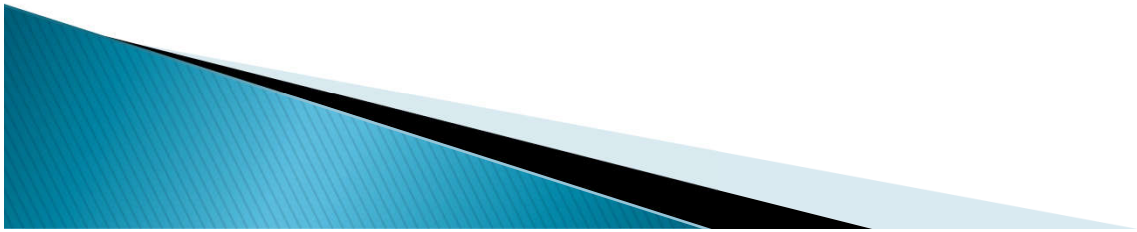
Software Engineering Observation 10.9

Using a superclass reference, we can polymorphically invoke any method declared in the superclass and its superclasses (e.g., class `Object`*). Using an interface reference, we can polymorphically invoke any method declared in the interface, its superinterfaces (one interface can extend another) and in class* `Object`*—a variable of an interface type must refer to an object to call methods, and all objects have the methods of class* `Object`*.*

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

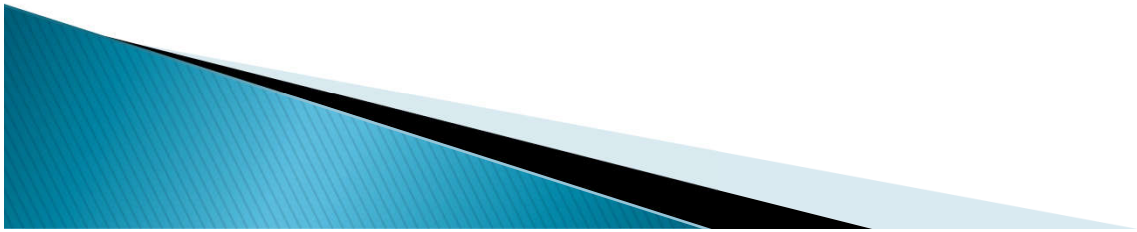
```
1  // Fig. 10.14: SalariedEmployee.java
2  // SalariedEmployee class extends Employee, which implements Payable.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
8     // four-argument constructor
9     public SalariedEmployee( String first, String last, 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
```

Fig. 10.14 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`. (Part 1 of 3.)



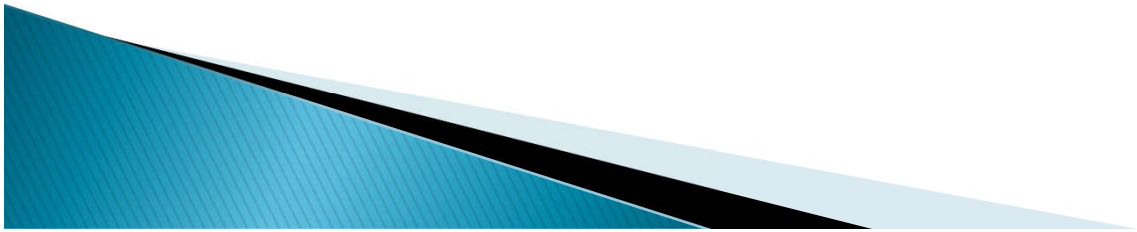
```
16      // set salary
17      public void setWeeklySalary( double salary )
18      {
19         if ( salary >= 0.0 )
20            baseSalary = salary;
21         else
22            throw new IllegalArgumentException(
23               "Weekly salary must be >= 0.0" );
24      } // end method setWeeklySalary
25
26      // return salary
27      public double getWeeklySalary()
28      {
29         return weeklySalary;
30      } // end method getWeeklySalary
31
```

Fig. 10.14 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`. (Part 2 of 3.)



```
32      // calculate earnings; implement interface Payable method that was
33      // abstract in superclass Employee
34      @Override
35      public double getPaymentAmount()
36      {
37         return getWeeklySalary();
38      } // end method getPaymentAmount
39
40      // return String representation of SalariedEmployee object
41      @Override
42      public String toString()
43      {
44         return String.format( "salaried employee: %s\n%s: $%,.2f",
45            super.toString(), "weekly salary", getWeeklySalary() );
46      } // end method toString
47   } // end class SalariedEmployee
```

Fig. 10.14 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`. (Part 3 of 3.)



10.7.5 Modifying Class SalariedEmployee for Use in the Payable Hierarchy (Cont.)

- Objects of any subclasses of a class that `implements` an interface can also be thought of as objects of the interface type.
- Thus, just as we can assign the reference of a `SalariedEmployee` object to a superclass `Employee` variable, we can assign the reference of a `SalariedEmployee` object to an interface `Payable` variable.
- `Invoice` implements `Payable`, so an `Invoice` object also *is a* `Payable` *object, and we can assign the reference of an* `Invoice` *object to a* `Payable` *variable.*

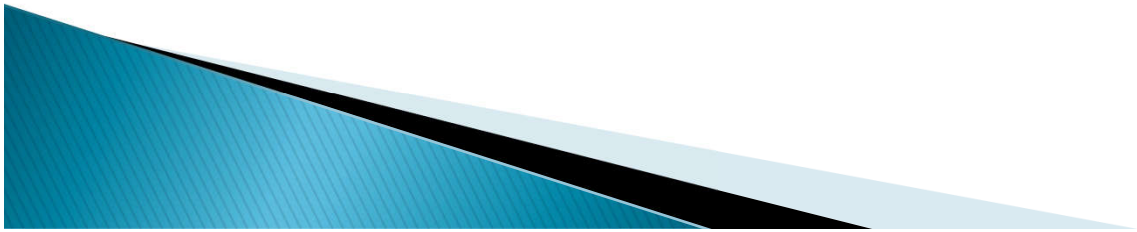
© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.7.6 Using Interface Payable to Process Invoices and Employees Polymorphically

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

```
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
```

Fig. 10.15 | `Payable` interface test program processing `Invoice`s and `Employee`s polymorphically. (Part 1 of 3.)

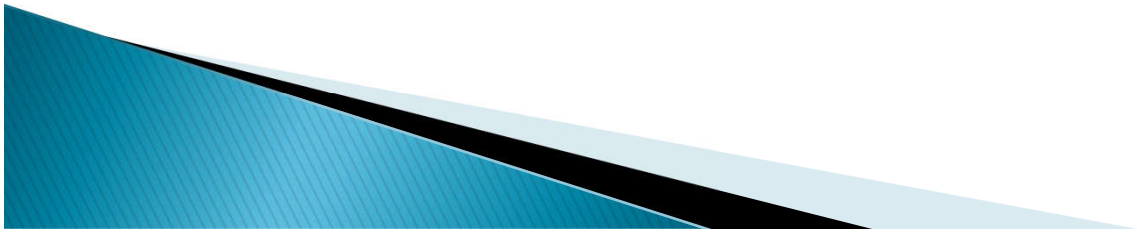


```
22         // generically process each element in array payableObjects
23         for ( Payable currentPayable : payableObjects )
24         {
25            // output currentPayable and its appropriate payment amount
26            System.out.printf( "%s \n%s: $%,.2f\n\n",
27               currentPayable.toString(),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

```
Invoices and Employees processed polymorphically:

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

Fig. 10.15 | `Payable` interface test program processing `Invoice`s and `Employee`s polymorphically. (Part 2 of 3.)

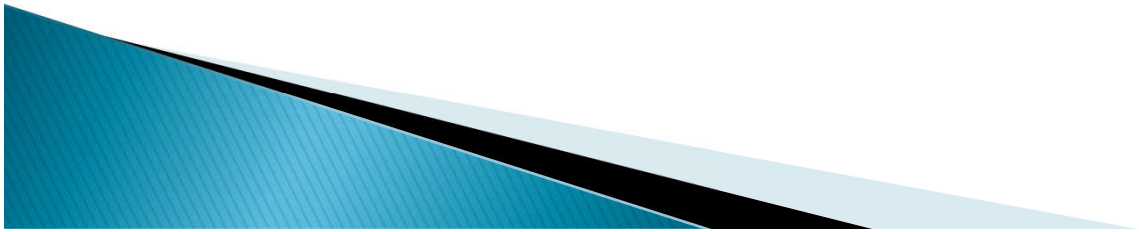


```
invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

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
```

Fig. 10.15 | `Payable` interface test program processing `Invoice`s and `Employee`s polymorphically. (Part 3 of 3.)



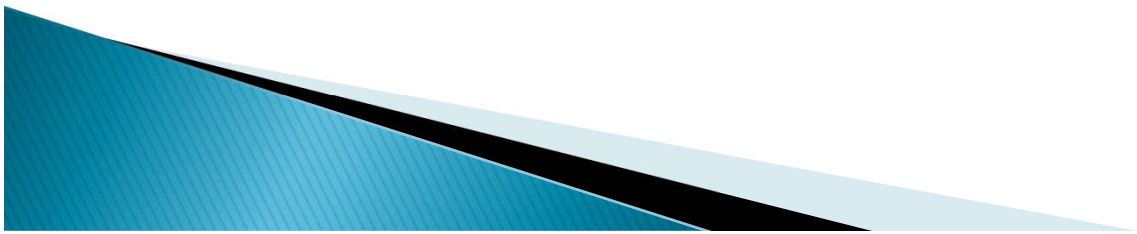
10.7.7 Common Interfaces of the Java API

- The Java API's interfaces enable you to use your own classes within the frameworks provided by Java, such as comparing objects of your own types and creating tasks that can execute concurrently with other tasks in the same program.
- Figure 10.16 presents a brief overview of a few of the more popular interfaces of the Java API that we use in *Java How to Program, Ninth Edition.*

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

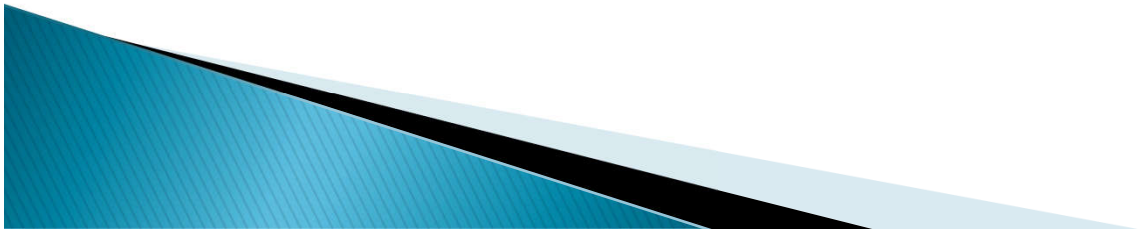
Interface	Description
`Comparable`	Java contains several comparison operators (e.g., <, <=, >, >=, ==, !=) that allow you to compare primitive values. However, these operators *cannot* be used to compare objects. Interface `Comparable` is used to allow objects of a class that `implements` the interface to be compared to one another. Interface `Comparable` is commonly used for ordering objects in a collection such as an array. We use `Comparable` in Chapter 20, Generic Collections, and Chapter 21, Generic Classes and Methods.
`Serializable`	An interface used 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 17, Files, Streams and Object Serialization, and Chapter 27, Networking.
`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 26, Multithreading). The interface contains one method, `run`, which describes the behavior of an object when executed.

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



Interface	Description
GUI event-listener interfaces	You work with graphical user interfaces (GUIs) every day. In your web browser, you might type the address of a website to visit, or you might click a button to return to a previous site. The browser responds to your interaction and performs the desired task. 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 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2, you'll learn how to build GUIs and event handlers that respond to user interactions. 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 14 and 25.

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



10.8 (Optional) GUI and Graphics Case Study: Drawing with Polymorphism

- Shape classes have many similarities.
- Using inheritance, we can "factor out" the common features from all three classes and place them in a single shape superclass.
- Then, using variables of the superclass type, we can manipulate objects of all three shape types polymorphically.
- Removing the redundancy in the code will result in a smaller, more flexible program that is easier to maintain.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

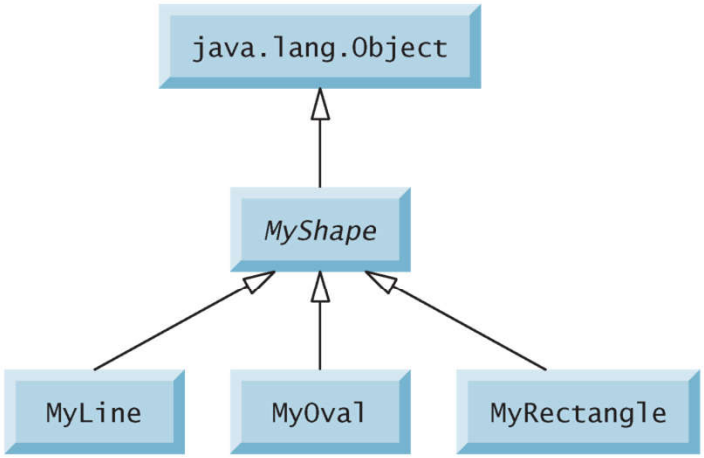


Fig. 10.17 | MyShape hierarchy.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

10.8 (Optional) GUI and Graphics Case Study: Drawing with Polymorphism (Cont.)

- Class `MyBoundedShape` can be used to factor out the common features of classes `MyOval` and `MyRectangle`.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.

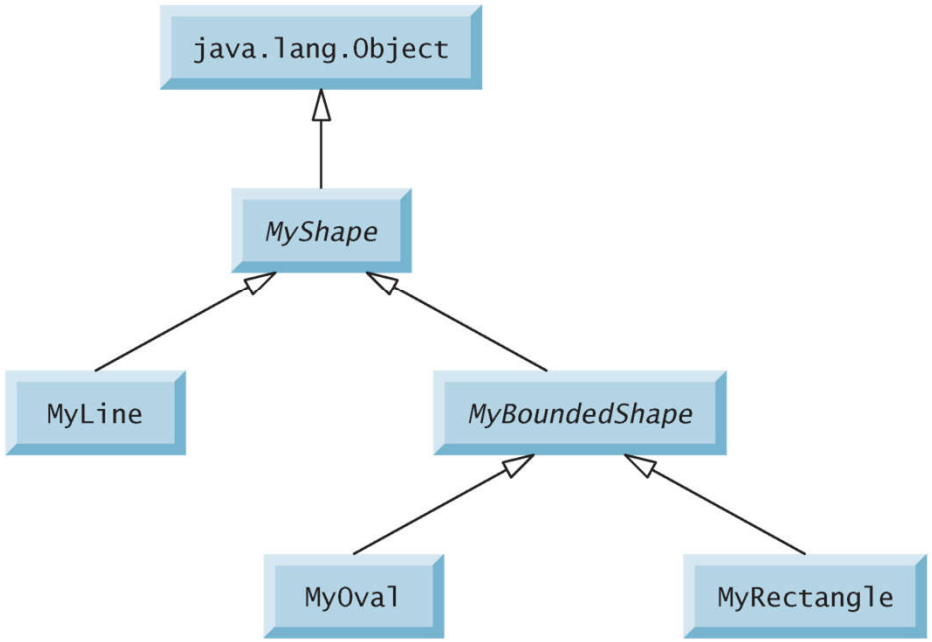


Fig. 10.18 | MyShape hierarchy with MyBoundedShape.

© Copyright 1992–2012 by Pearson Education, Inc. All Rights Reserved.