Object-Oriented Programming: Polymorphism

Introduction

- Polymorphism
  - Enables you to "program in the general" rather than "program in the specific."
  - Polymorphism enables you to write programs that process objects that share the same superclass as if they're all objects of the superclass; this can simplify programming.

Introduction (Cont.)

- Example: Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the three types of animals under investigation.
  - Each class extends superclass `Animal`, which contains a method `move` and maintains an animal's current location as *x-y* coordinates. Each subclass implements method `move`.
  - A program maintains an `Animal` array containing references to objects of the various `Animal` subclasses. To simulate the animals' movements, the program sends each object the same message once per second—namely, `move`.

Introduction (Cont.)

- Each specific type of `Animal` responds to a `move` message in a unique way:
  - a `Fish` might swim three feet
  - a `Frog` might jump five feet
  - a `Bird` might fly ten feet.
- The program issues the same message (i.e., `move`) to each animal object, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement.
- Relying on each object to know how to "do the right thing" in response to the same method call is the key concept of polymorphism.
- The same message sent to a variety of objects has "many forms" of results—hence the term polymorphism.

Introduction (Cont.)

- With polymorphism, we can design and implement systems that are easily *extensible*
  - New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically.
  - The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that we add to the hierarchy.

Introduction (Cont.)

- Once a class implements an interface, all objects of that class have an *is-a* relationship with the interface type, and all objects of the class are guaranteed to provide the functionality described by the interface.
- This is true of all subclasses of that class as well.
- Interfaces are particularly useful for assigning common functionality to possibly unrelated classes.
  - Allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to all of the interface method calls.

Introduction (Cont.)

- An interface describes a set of methods that can be called on an object, but does not provide concrete implementations for all the methods.
- You can declare classes that implement (i.e., provide concrete implementations for the methods of) one or more interfaces.
- Each interface method must be declared in all the classes that explicitly implement the interface.

Polymorphism Examples

- Example: Quadrilaterals
  - If `Rectangle` is derived from `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral`.
  - Any operation that can be performed on a `Quadrilateral` can also be performed on a `Rectangle`.
  - These operations can also be performed on other `Quadrilaterals`, such as `Squares`, `Parallelograms` and `Trapezoids`.
  - Polymorphism occurs when a program invokes a method through a superclass `Quadrilateral` variable—at execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.

Polymorphism Examples (Cont.)

- Example: Space Objects in a Video Game
  - A video game manipulates objects of classes `Martian`, `Venusian`, `Plutonian`, `SpaceShip` and `LaserBeam`. Each inherits from `SpaceObject` and overrides its `draw` method.
  - A screen manager maintains a collection of references to objects of the various classes and periodically sends each object the same message—namely, `draw`.
  - Each object responds in a unique way.
    - A `Martian` object might draw itself in red with green eyes and the appropriate number of antennae.
    - A `SpaceShip` object might draw itself as a bright silver flying saucer.
    - A `LaserBeam` object might draw itself as a bright red beam across the screen.
  - The same message (in this case, `draw`) sent to a variety of objects has "many forms" of results.

Polymorphism Examples (Cont.)

- A screen manager might use polymorphism to facilitate adding new classes to a system with minimal modifications to the system's code.
- To add new objects to our video game:
  - Build a class that extends `SpaceObject` and provides its own `draw` method implementation.
  - When objects of that class appear in the `SpaceObject` collection, the screen manager code invokes method `draw`, exactly as it does for every other object in the collection, regardless of its type.
  - So the new objects simply "plug right in" without any modification of the screen manager code by the programmer.

Software Engineering Observation 10.1

Polymorphism enables you to deal in generalities and let the execution-time environment handle the specifics. You can command objects to behave in manners appropriate to those objects, without knowing their types (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 modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

Demonstrating Polymorphic Behavior

- In the next example, we aim a superclass reference at a subclass object.
  - Invoking a method on a subclass object via a superclass reference invokes the subclass functionality
  - The type of the referenced object, not the type of the variable, determines which method is called
- This example demonstrates that an object of a subclass can be treated as an object of its superclass, enabling various interesting manipulations.
- A program can create an array of superclass variables that refer to objects of many subclass types.
  - Allowed because each subclass object *is an* object of its superclass.

Demonstrating Polymorphic Behavior (Cont.)

- A superclass object cannot be treated as a subclass object, because a superclass object is *not* an object of any of its subclasses.
- The *is-a* relationship applies only up the hierarchy from a subclass to its direct (and indirect) superclasses, and not down the hierarchy.
- The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if you explicitly cast the superclass reference to the subclass type
  - A technique known as downcasting that enables a program to invoke subclass methods that are not in the superclass.

```
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       CommissionEmployee commissionEmployee = new CommissionEmployee(
11          "Sue", "Jones", "222-22-2222", 10000, .06 );
12
13       // assign subclass reference to subclass variable
14       BasePlusCommissionEmployee basePlusCommissionEmployee =
15          new BasePlusCommissionEmployee(
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 CommissionEmployee's toString with superclass reference ",
21          "to superclass object", commissionEmployee.toString() );
22
```

Variable refers to a `CommissionEmployee` object, so that class's `toString` method is called

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 1 of 3.)


```
23       // invoke toString on subclass object using subclass variable
24       System.out.printf( "%s %s:\n\n%s\n\n",
25          "Call BasePlusCommissionEmployee's toString with subclass",
26          "reference to subclass object",
27          basePlusCommissionEmployee.toString() );
28
29       // invoke toString on subclass object using superclass variable
30       CommissionEmployee commissionEmployee2 =
31          basePlusCommissionEmployee;
32       System.out.printf( "%s %s:\n\n%s\n",
33          "Call BasePlusCommissionEmployee's toString with superclass",
34          "reference to subclass object", commissionEmployee2.toString() );
35    } // end main
36 } // end class PolymorphismTest
```

Variable refers to a `BasePlus-CommissionEmployee` object, so that class's `toString` method is called

Variable refers to a `BasePlus-CommissionEmployee` object, so that class's `toString` method is called

```
Call CommissionEmployee'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
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 2 of 3.)

```
Call BasePlusCommissionEmployee'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 BasePlusCommissionEmployee'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
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 3 of 3.)

Demonstrating Polymorphic Behavior (Cont.)

- When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.
  - The Java compiler allows this "crossover" because an object of a subclass *is an object of its superclass (but not vice versa)*.
- When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type.
  - If that class contains the proper method declaration (or inherits one), the call is compiled.
- At execution time, the type of the object to which the variable refers determines the actual method to use.
  - This process is called dynamic binding.

Abstract Classes and Methods

- Abstract classes
  - Sometimes it's useful to declare classes for which you never intend to create objects.
  - Used only as superclasses in inheritance hierarchies, so they are sometimes called abstract superclasses.
  - Cannot be used to instantiate objects—abstract classes are incomplete.
  - Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract.
- An abstract class provides a superclass from which other classes can inherit and thus share a common design.

Abstract Classes and Methods (Cont.)

- Classes that can be used to instantiate objects are called concrete classes.
- Such classes provide implementations of every method they declare (some of the implementations can be inherited).
- Abstract superclasses are too general to create real objects—they specify only what is common among subclasses.
- Concrete classes provide the specifics that make it reasonable to instantiate objects.
- Not all hierarchies contain abstract classes.

Abstract Classes and Methods (Cont.)

- Programmers often write client code that uses only abstract superclass types to reduce client code's dependencies on a range of subclass types.
  - You can write a method with a parameter of an abstract superclass type.
  - When called, such a method can receive an object of any concrete class that directly or indirectly extends the superclass specified as the parameter's type.
- Abstract classes sometimes constitute several levels of a hierarchy.

Abstract Classes and Methods (Cont.)

- You make a class abstract by declaring it with keyword `abstract`.
- An abstract class normally contains one or more abstract methods.
  - An abstract method is one with keyword `abstract` in its declara-tion, as in

```
public abstract void draw(); // abstract method
```

- Abstract methods do not provide implementations.
- A class that contains abstract methods must be an abstract class even if that class contains some concrete (nonabstract) methods.
- Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the superclass's abstract methods.
- Constructors and `static` methods cannot be declared `abstract`.

Software Engineering Observation 10.3

An abstract class declares common attributes and behaviors (both abstract and concrete) of the various classes in a class hierarchy. An abstract class typically contains one or more abstract methods that subclasses must override if they are to be concrete. The instance variables and concrete methods of an abstract class are subject to the normal rules of inheritance.

Common Programming Error 10.1

Attempting to instantiate an object of an abstract class is a compilation error.

Common Programming Error 10.2

Failure to implement a superclass's abstract methods in a subclass is a compilation error unless the subclass is also declared `abstract`.

Abstract Classes and Methods (Cont.)

- Cannot instantiate objects of abstract superclasses, but you can use abstract superclasses to declare variables
  - These can hold references to objects of any concrete class derived from those abstract superclasses.
  - Programs typically use such variables to manipulate subclass objects polymorphically.
- Can use abstract superclass names to invoke `static` methods declared in those abstract superclasses.

Abstract Classes and Methods (Cont.)

- Polymorphism is particularly effective for implementing so-called *layered software systems*.
- Example: Operating systems and device drivers.
  - Commands to read or write data from and to devices may have a certain uniformity.
  - Device drivers control all communication between the operating system and the devices.
  - A write message sent to a device-driver object is interpreted in the context of that driver and how it manipulates devices of a specific type.
  - The write call itself really is no different from the write to any other device in the system—place some number of bytes from memory onto that device.

Abstract Classes and Methods (Cont.)

- An object-oriented operating system might use an abstract superclass to provide an "interface" appropriate for all device drivers.
  - Subclasses are formed that all behave similarly.
  - The device-driver methods are declared as abstract methods in the abstract superclass.
  - The implementations of these abstract methods are provided in the subclasses that correspond to the specific types of device drivers.
- New devices are always being developed.
  - When you buy a new device, it comes with a device driver provided by the device vendor and is immediately operational after you connect it and install the driver.
- This is another elegant example of how polymorphism makes systems extensible.

Case Study: Payroll System Using Polymorphism

- Use an abstract method and polymorphism to perform payroll calculations based on the type of inheritance hierarchy headed by an employee.
- Enhanced employee inheritance hierarchy requirements:
  - A company pays its employees on a weekly basis. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay (i.e., 1.5 times their hourly salary rate) for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales and base-salaried commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries. The company wants to write a Java application that performs its payroll calculations polymorphically.

Case Study: Payroll System Using Polymorphism (Cont.)

- `abstract` class `Employee` represents the general concept of an employee.
- Subclasses: `SalariedEmployee`, `CommissionEmployee` , `HourlyEmployee` and `BasePlusCommissionEmployee` (an indirect subclass)
- Fig. 10.2 shows the inheritance hierarchy for our polymorphic employee-payroll application.
- Abstract class names are italicized in the UML.

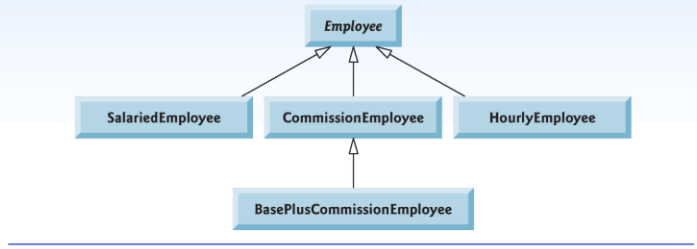

Fig. 10.2 | `Employee` hierarchy UML class diagram.

Case Study: Payroll System Using Polymorphism (Cont.)

- Abstract superclass `Employee` declares the "interface" to the hierarchy—that is, the set of methods that a program can invoke on all `Employee` objects.
  - We use the term "interface" here in a general sense to refer to the various ways programs can communicate with objects of any `Employee` subclass.
- Each employee has a first name, a last name and a social security number defined in abstract superclass `Employee`.

Abstract Superclass `Employee`

- Class `Employee` (Fig. 10.4) provides methods `earnings` and `toString`, in addition to the *get* and *set* methods that manipulate `Employee`'s instance variables.
- An `earnings` method applies to all employees, but each earnings calculation depends on the employee's class.
  - An `abstract` method—there is not enough information to determine what amount `earnings` should return.
  - Each subclass overrides `earnings` with an appropriate implementation.
- Iterate through the array of `Employee`s and call method `earnings` for each `Employee` subclass object.
  - Method calls processed polymorphically.

Abstract Superclass `Employee` (Cont.)

- The diagram in Fig. 10.3 shows each of the five classes in the hierarchy down the left side and methods `earnings` and `toString` across the top.
- For each class, the diagram shows the desired results of each method.
- Declaring the `earnings` method `abstract` indicates that each concrete subclass must provide an appropriate `earnings` implementation and that a program will be able to use superclass `Employee` variables to invoke method `earnings` polymorphically for any type of `Employee`.

	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
	else 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.


```
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
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first; // should validate
22    } // end method setFirstName
23
```

Fig. 10.4 | `Employee` abstract superclass. (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. 10.4 | `Employee` abstract superclass. (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     // abstract method overridden by concrete subclasses
63     public abstract double earnings(); // no implementation here
64  } // end abstract class Employee
```

This method must be overridden in subclasses to make them concrete

Fig. 10.4 | `Employee` abstract superclass. (Part 3 of 3.)

```
1  // Fig. 10.5: SalariedEmployee.java
2  // SalariedEmployee concrete class extends abstract class 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
```

Fig. 10.5 | `SalariedEmployee` concrete class extends `abstract` class `Employee`. (Part 1 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    @Override
30    public double earnings()
31    {
32       return getWeeklySalary();
33    } // end method earnings
34
35    // return String representation of SalariedEmployee object
36    @Override
37    public String toString()
38    {
39       return String.format( "salaried employee: %s\n%s: $%,.2f",
40          super.toString(), "weekly salary", getWeeklySalary() );
41    } // end method toString
42 } // end class SalariedEmployee
```

Overriding `earnings` makes this class concrete

Overriding `toString` provides customized `String` representation for this class

Fig. 10.5 | `SalariedEmployee` concrete class extends `abstract` class `Employee`. (Part 2 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
```

Fig. 10.6 | `HourlyEmployee` class derived from `Employee`. (Part 1 of 3.)


```
24    // return wage
25    public double getWage()
26    {
27       return wage;
28    } // end method getWage
29
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
```

Fig. 10.6 | `HourlyEmployee` class derived from `Employee`. (Part 2 of 3.)

```
43     // calculate earnings; override abstract method earnings in Employee
44     @Override
45     public double earnings()
46     {
47        if ( getHours() <= 40 ) // no overtime
48           return getWage() * getHours();
49        else
50           return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
51     } // end method earnings
52
53     // return String representation of HourlyEmployee object
54     @Override
55     public String toString()
56     {
57        return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
58           super.toString(), "hourly wage", getWage(),
59           "hours worked", getHours() );
60     } // end method toString
61  } // end class HourlyEmployee
```

Overriding `earnings` makes this class concrete

Overriding `toString` provides customized `String` representation for this class

Fig. 10.6 | `HourlyEmployee` class derived from `Employee`. (Part 3 of 3.)


```
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
```

Fig. 10.7 | `CommissionEmployee` class derived from `Employee`. (Part 1 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
42    // calculate earnings; override abstract method earnings in Employee
43    @Override
44    public double earnings()
45    {
46       return getCommissionRate() * getGrossSales();
47    } // end method earnings
```

Overriding `earnings` makes this class concrete

Fig. 10.7 | `CommissionEmployee` class derived from `Employee`. (Part 2 of 3.)


```
48
49    // return String representation of CommissionEmployee object
50    @Override
51    public String toString()
52    {
53       return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
54          "commission employee", super.toString(),
55          "gross sales", getGrossSales(),
56          "commission rate", getCommissionRate() );
57    } // end method toString
58 } // end class CommissionEmployee
```

Overriding `toString` provides customized `String` representation for this class

Fig. 10.7 | `CommissionEmployee` class derived from `Employee`. (Part 3 of 3.)

```java
// Fig. 10.8: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee class extends CommissionEmployee.

public class BasePlusCommissionEmployee extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
   } // end method setBaseSalary
```

Fig. 10.8 | `BasePlusCommissionEmployee` class extends `CommissionEmployee`. (Part 1 of 2.)


```java
   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings; override method earnings in CommissionEmployee
   @Override
   public double earnings()
   {
      return getBaseSalary() + super.earnings();
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee object
   @Override
   public String toString()
   {
      return String.format( "%s %s; %s: $%,.2f",
         "base-salaried", super.toString(),
         "base salary", getBaseSalary() );
   } // end method toString
} // end class BasePlusCommissionEmployee
```

If we do not override `earnings` in this class, we inherit the version in from superclass `CommissionEmployee` and this class is still a concrete class

Overriding `toString` provides customized `String` representation for this class

Fig. 10.8 | `BasePlusCommissionEmployee` class extends `CommissionEmployee`. (Part 2 of 2.)

Polymorphic Processing, Operator `instanceof` and Downcasting

- Fig. 10.9 creates an object of each of the four concrete.
  - Manipulates these objects nonpolymorphically, via variables of each object's own type, then polymorphically, using an array of `Employee` variables.
- While processing the objects polymorphically, the program increases the base salary of each `BasePlusCommissionEmployee` by 10%
  - Requires determining the object's type at execution time.
- Finally, the program polymorphically determines and outputs the type of each object in the `Employee` array.


```
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
22       System.out.printf( "%s\n%s: $%,.2f\n\n",
23          salariedEmployee, "earned", salariedEmployee.earnings() );
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 1 of 6.)

```
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
```

Does not create Employee objects—just variables that can refer to objects of Employee subclasses

Aim each Employee variable at an object of an Employee subclass

Polymorphically invokes toString

Fig. 10.9 | Employee hierarchy test program. (Part 2 of 6.)


```
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               employee.setBaseSalary( 1.10 * employee.getBaseSalary() );
57
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
```

Is currentEmployee a BasePlus-CommissionEmployee?

This downcast works because currentEmployee is a BasePlus-CommissionEmployee

Polymorphically invokes earnings

Every object in Java knows its own type

Fig. 10.9 | Employee hierarchy test program. (Part 3 of 6.)

```
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
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 4 of 6.)


```
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
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 5 of 6.)

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

Fig. 10.9 | `Employee` hierarchy test program. (Part 6 of 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.

Common Programming Error 10.3

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

Common Programming Error 10.4

When downcasting an object, a `ClassCastException` *occurs if at execution time the object does not have an* is-a *relationship with the type specified in the cast operator. A reference can be cast only to its own type or to the type of one of its superclasses.*

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.

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.

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.

`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.

`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.

Common Programming Error 10.5

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

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.*

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.

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).

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.

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`.

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`."

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`.

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.

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.

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.

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.*

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.

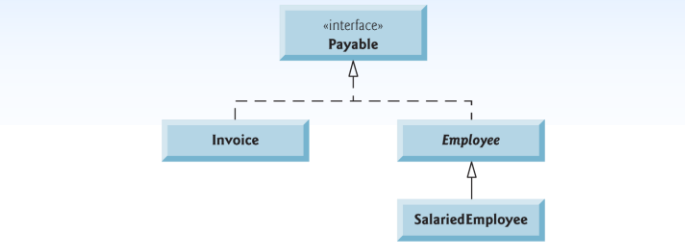

Fig. 10.10 | `Payable` interface hierarchy UML class diagram.

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`.

```
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.

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, ...
```

```
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
```

Class extends `Object` (implicitly) and implements interface `Payable`

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        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
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
```

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


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

Providing an implementation of the interface's method(s) makes this class concrete

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.*

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.

```
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
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first; // should validate
22    } // end method setFirstName
23
```

Abstract class extends `Object` (implicitly) and implements interface `Payable`

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
```

We don't implement the interface's method, so this class must be declared `abstract`

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

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`.

```
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
```

Fig. 10.14 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`. (Part 1 of 2.)


```
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    @Override
31    public double getPaymentAmount()
32    {
33       return getWeeklySalary();
34    } // end method getPaymentAmount
35
36    // return String representation of SalariedEmployee object
37    @Override
38    public String toString()
39    {
40       return String.format( "salaried employee: %s\n%s: $%,.2f",
41          super.toString(), "weekly salary", getWeeklySalary() );
42    } // end method toString
43 } // end class SalariedEmployee
```

Providing an implementation of the interface's method(s) makes this class concrete

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

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.*

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.

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`*.*

Using Interface `Payable` to Process `Invoices` and `Employees` Polymorphically

```
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
```

Creates an array of four `Payable` variables

Aim each `Payable` variable at an object of a class that implement the `Payable` interface

Fig. 10.15 | `Payable` interface test program processing `Invoices` and `Employees` 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
```

Fig. 10.15 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 2 of 3.)


```
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
```

Fig. 10.15 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 3 of 3.)

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, Eighth Edition*.

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.)