Object-Oriented Programming: Polymorphism
A Motivating Example

• Employee as an abstract superclass.
• Lots of different types of employees (well, 4).
• Executing the same code on all different types of employees and letting the run-time system figure out which type of employee is being referenced.
Fig. 10.2 | Employee hierarchy UML class diagram.
// Fig. 10.9: PayrollSystemTest.java
// Employee hierarchy test program

class PayrollSystemTest
{
    public static void main( String args[] )
    {
        // create subclass objects
        SalariedEmployee salariedEmployee =
            new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
        HourlyEmployee hourlyEmployee =
            new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
        CommissionEmployee commissionEmployee =
            new CommissionEmployee( "Sue", "Jones", "333-33-3333", 10000, .06 );
        BasePlusCommissionEmployee basePlusCommissionEmployee =
            new BasePlusCommissionEmployee( "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
        System.out.println( "Employees processed individually:\n" );
System.out.printf("%n%s: $%.2f%n", salariedEmployee, "earned", salariedEmployee.earnings());
System.out.printf("%n%s: $%.2f%n", hourlyEmployee, "earned", hourlyEmployee.earnings());
System.out.printf("%n%s: $%.2f%n", commissionEmployee, "earned", commissionEmployee.earnings());
System.out.printf("%n%s: $%.2f%n", basePlusCommissionEmployee, "earned", basePlusCommissionEmployee.earnings());

// create four-element Employee array
Employee employees[] = new Employee[4];

// initialize array with Employees
employees[0] = salariedEmployee;
employees[1] = hourlyEmployee;
employees[2] = commissionEmployee;

System.out.println("Employees processed polymorphically:
");

// generically process each element in array employees
for (currentEmployee = 0; currentEmployee < 4; currentEmployee++)
{
    System.out.println(employees[currentEmployee].toString); // cool!
}
If the `currentEmployee` variable points to a `BasePlusCommissionEmployee` object

- **PayrollSystem**

  Downcast `currentEmployee` to a `BasePlusCommissionEmployee` reference

  Give `BasePlusCommissionEmployee`s a 10% base salary bonus

  Polymorphically call `earnings` method

  Call `getClass` and `getName` methods to display each `Employee` subclass object’s class name
Employees processed individually:

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

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

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

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

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

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

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

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

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

Same results as when the employees were processed individually

• PayrollSystemTest
• .java

• (5 of 5)

Base salary is increased by 10%

Each employee’s type is displayed
10.1 Introduction

• Polymorphism
  – Enables “programming in the general”
  – The same invocation can produce “many forms” of results

• Interfaces
  – Implemented by classes to assign common functionality to possibly unrelated classes
10.2 Polymorphism Examples

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

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

• A subclass reference can be aimed at a superclass object only if the object is downcasted
/* Fig. 10.1: PolymorphismTest.java */
/* Assigning superclass and subclass references to superclass and */
/* subclass variables. */

public class PolymorphismTest {
    public static void main(String args[])
    {
        // assign superclass reference to superclass variable
        CommissionEmployee3 commissionEmployee = new CommissionEmployee3("Sue", "Jones", "222-22-2222", 10000, .06);

        // assign subclass reference to subclass variable

        // invoke toString on superclass object using superclass variable
        System.out.printf("%s %s:

%s

", "Call CommissionEmployee3's toString with superclass reference ", "to superclass object", commissionEmployee.toString());

        // invoke toString on subclass object using subclass variable
        System.out.printf("%s %s:

%s

", "Call BasePlusCommissionEmployee4's toString with subclass", "reference to subclass object", basePlusCommissionEmployee.toString());
// invoke toString on subclass object using superclass variable

CommissionEmployee3 commissionEmployee2 = basePlusCommissionEmployee;
System.out.printf( "%s %s:

%s
", "Call BasePlusCommissionEmployee4's toString with superclass",
"reference to subclass object", commissionEmployee2.toString() );

} // end main

} // end class PolymorphismTest

Call CommissionEmployee3's toString with superclass object:
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee4's toString with subclass reference to subclass object:
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee4's toString with superclass reference to subclass object:
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Assign a reference to a basePlusCommissionEmployee object to a CommissionEmployee3 variable

Polymorphically call basePlusCommissionEmployee's toString method
10.4 Abstract Classes and Methods

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

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

• Next… the whole Employee example…
Fig. 10.2 | Employee hierarchy UML class diagram.
Declare **abstract** class **Employee**

Attributes common to all employees
// set first name
public void setFirstName(String first)
{
    firstName = first;
} // end method setFirstName

// return first name
public String getFirstName()
{
    return firstName;
} // end method getFirstName

// set last name
public void setLastName(String last)
{
    lastName = last;
} // end method setLastName

// return last name
public String getLastName()
{
    return lastName;
} // end method getLastName
```java
// set social security number
public void setSocialSecurityNumber( String ssn )
{
    socialSecurityNumber = ssn; // should validate
} // end method setSocialSecurityNumber

// return social security number
public String getSocialSecurityNumber()
{
    return socialSecurityNumber;
} // end method getSocialSecurityNumber

// return String representation of Employee object
public String toString()
{
    return String.format( "%s %s
social security number: %s",
        getFirstName(), getLastName(), getSocialSecurityNumber() );
} // end method toString

// abstract method overridden by subclasses
public abstract double earnings(); // no implementation here
} // end abstract class Employee
```

**abstract method earnings**

has no implementation
// Fig. 10.5: SalariedEmployee.java
// SalariedEmployee class extends Employee.

public class SalariedEmployee extends Employee {
    private double weeklySalary;

    // four-argument constructor
    public SalariedEmployee( String first, String last, String ssn, double salary )
    {
        super( first, last, ssn ); // pass to Employee constructor
        setWeeklySalary( salary ); // validate and store salary
    } // end four-argument SalariedEmployee constructor

    // set salary
    public void setWeeklySalary( double salary )
    {
        weeklySalary = salary < 0.0 ? 0.0 : salary;
    } // end method setWeeklySalary
} // end class SalariedEmployee
// return salary
public double getWeeklySalary()
{
    return weeklySalary;
} // end method getWeeklySalary

// calculate earnings; override abstract method earnings in Employee
public double earnings()
{
    return getWeeklySalary();
} // end method earnings

// return String representation of SalariedEmployee object
public String toString()
{
    return String.format( "salaried employee: %s
weekly salary: $%,.2f", super.toString(), "weekly salary", getWeeklySalary() );
} // end method toString

// end class SalariedEmployee

Override properties so SalariedEmployee can be concrete

Override earnings method so SalariedEmployee can be concrete

Override toString method

Call superclass’s version of toString
```java
public class HourlyEmployee extends Employee {

    private double wage; // wage per hour
    private double hours; // hours worked for week

    // five-argument constructor
    public HourlyEmployee( String first, String last, String ssn,
                              double hourlyWage, double hoursWorked )
    {
        super( first, last, ssn );
        setWage( hourlyWage ); // validate hourly wage
        setHours( hoursWorked ); // validate hours worked
    } // end five-argument HourlyEmployee constructor

    // set wage
    public void setWage( double hourlyWage )
    {
        wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
    } // end method setWage

    // return wage
    public double getWage()
    {
        return wage;
    } // end method getWage
```
// set hours worked
public void setHours( double hoursWorked )
{
    hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
        hoursWorked : 0.0;
} // end method setHours

// return hours worked
public double getHours()
{
    return hours;
} // end method getHours

// calculate earnings; override abstract method earnings in Employee
public double earnings()
{
    if ( getHours() <= 40 ) // no overtime
        return getWage() * getHours();
    else
        return 40 * getWage() + ( gethours() - 40 ) * getWage() * 1.5;
} // end method earnings

// return String representation of HourlyEmployee object
public String toString()
{
    return String.format( "hourly employee: %s
%s: $%,.2f; %s: %,.2f",
        super.toString(), "hourly wage", getWage(),
        "hours worked", getHours() );
} // end method toString

} // end class HourlyEmployee
public class CommissionEmployee extends Employee {

    private double grossSales; // gross weekly sales
    private double commissionRate; // commission percentage

    // five-argument constructor
    public CommissionEmployee( String first, String last, String ssn,
            double sales, double rate )
    {
        super( first, last, ssn );
        setGrossSales( sales );
        setCommissionRate( rate );
    } // end five-argument CommissionEmployee constructor

    // set commission rate
    public void setCommissionRate( double rate )
    {
        commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
    } // end method setCommissionRate

// return commission rate
public double getCommissionRate()
{
    return commissionRate;
} // end method getCommissionRate

// set gross sales amount
public void setGrossSales( double sales )
{
    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
} // end method setGrossSales

// return gross sales amount
public double getGrossSales()
{
    return grossSales;
} // end method getGrossSales
// calculate earnings; override abstract method earnings in Employee
public double earnings()
{
    return getCommissionRate() * getGrossSales();
} // end method earnings

// return String representation of CommissionEmployee object
public String toString()
{
    return String.format("%s: %s\n%s: $%,.2f; %s: %.2f",
                        "commission employee", super.toString(),
                        "gross sales", getGrossSales(),
                        "commission rate", getCommissionRate());
} // end method toString

} // end class CommissionEmployee

Override earnings method so CommissionEmployee can be concrete

Override toString method

Call superclass’s toString method
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

    // set base salary
    public void setBaseSalary( double sales )
    {
        // Call superclass constructor
        // Validate and set base salary value
    } // end method setBaseSalary

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

    // six-argument BasePlusCommissionEmployee class
// return base salary
public double getBaseSalary()
{
    return baseSalary;
} // end method getBaseSalary

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

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

} // end class BasePlusCommissionEmployee
// Fig. 10.9: PayrollSystemTest.java
// Employee hierarchy test program

public class PayrollSystemTest

public static void main(String args[])
{
    // create subclass objects
    SalariedEmployee salariedEmployee =
        new SalariedEmployee("John", "Smith", "111-11-1111", 800.00);
    HourlyEmployee hourlyEmployee =
        new HourlyEmployee("Karen", "Price", "222-22-2222", 16.75, 40);
    CommissionEmployee commissionEmployee =
        new CommissionEmployee("Sue", "Jones", "333-33-3333", 10000, .06);
    BasePlusCommissionEmployee basePlusCommissionEmployee =
        new BasePlusCommissionEmployee("Bob", "Lewis", "444-44-4444", 5000, .04, 300);

    System.out.println("Employees processed individually:\n");
System.out.printf("%s: $%,.2f\n\n", salariedEmployee, "earned", salariedEmployee.earnings());

System.out.printf("%s: $%,.2f\n\n", hourlyEmployee, "earned", hourlyEmployee.earnings());

System.out.printf("%s: $%,.2f\n\n", commissionEmployee, "earned", commissionEmployee.earnings());

System.out.printf("%s: $%,.2f\n\n", basePlusCommissionEmployee, "earned", basePlusCommissionEmployee.earnings());

// create four-element Employee array
Employee[] employees = new Employee[4];

// initialize array with Employees
employees[0] = salariedEmployee;
employees[1] = hourlyEmployee;
employees[2] = commissionEmployee;

System.out.println("Employees processed polymorphically:\n");

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

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

Downcast `currentEmployee` to a `BasePlusCommissionEmployee` reference

Give `BasePlusCommissionEmployee` a 10% base salary bonus

Polymorphically call `earnings` method

Call `getClass` and `getName` methods to display each `Employee` subclass object’s class name
Employees processed individually:

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

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

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

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

- **Salaried Employee:** John Smith
  - Social security number: 111-11-1111
  - Weekly salary: $800.00
  - Earned $800.00

- **Hourly Employee:** Karen Price
  - Social security number: 222-22-2222
  - Hourly wage: $16.75; hours worked: 40.00
  - Earned $670.00

- **Commission Employee:** Sue Jones
  - Social security number: 333-33-3333
  - Gross sales: $10,000.00; commission rate: 0.06
  - Earned $600.00

- **Base-Salaried Commission Employee:** Bob Lewis
  - Social security number: 444-44-4444
  - Gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
  - New base salary with 10% increase is: $330.00
  - Earned $530.00

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

Same results as when the employees were processed individually

Base salary is increased by 10%

Each employee’s type is displayed
10.5.6 Demonstrating Polymorphic Processing, Operator `instanceof` and Downcasting (Cont.)

- **Downcasting**
  - Convert a reference to a superclass to a reference to a subclass
  - Allowed only if the object has an *is-a* relationship with the subclass

- **`getClass` method**
  - Inherited from `Object`
  - Returns an object of type `Class`

- **`getName` method of class `Class`**
  - Returns the class’s name
10.5.7 Summary of the Allowed Assignments Between Superclass and Subclass Variables

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

**final methods**

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

**final classes**

- Cannot be extended by a subclass
- All methods in a `final` class are implicitly `final`
In the Java API, the vast majority of classes are not declared `final`. This enables inheritance and polymorphism—the fundamental capabilities of object-oriented programming. However, in some cases, it is important to declare classes `final`—typically for security reasons.
10.7 Case Study: Creating and Using Interfaces

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

• **Payable interface**
  - Contains method `getPaymentAmount`
  - Is implemented by the Invoice and Employee classes

• **UML representation of interfaces**
  - Interfaces are distinguished from classes by placing the word “interface” in guillemets (« and ») above the interface name
  - The relationship between a class and an interface is known as realization
    - A class “realizes” the method of an interface
Fig. 10.10  Payable interface hierarchy UML class diagram.
Declare interface **Payable**

```
public interface Payable {
    double getPaymentAmount(); // calculate payment; no implementation
}
```

Declare `getPaymentAmount` method which is implicitly **public** and **abstract**
// Fig. 10.12: Invoice.java
// Invoice class implements Payable.

public class Invoice implements Payable {
    private String partNumber;
    private String partDescription;
    private int quantity;
    private double pricePerItem

    // four-argument constructor
    public Invoice( String part, String description, int count,
            double price ) {
        partNumber = part;
        partDescription = description;
        setQuantity( count ); // validate and store quantity
        setPricePerItem( price ); // validate and store price per item
    } // end four-argument Invoice constructor

    // set part number
    public void setPartNumber( String part ) {
        partNumber = part;
    } // end method setPartNumber
// get part number
public String getPartNumber()
{
    return partNumber;
} // end method getPartNumber

// set description
public void setPartDescription( String description )
{
    partDescription = description;
} // end method setPartDescription

// get description
public String getPartDescription()
{
    return partDescription;
} // end method getPartDescription

// set quantity
public void setQuantity( int count )
{
    quantity = ( count < 0 ) ? 0 : count; // quantity cannot be negative
} // end method setQuantity

// get quantity
public int getQuantity()
{
    return quantity;
} // end method getQuantity
// set price per item
public void setPricePerItem( double price )
{
    pricePerItem = ( price < 0.0 ) ? 0.0 : price; // validate price
} // end method setPricePerItem

// get price per item
public double getPricePerItem()
{
    return pricePerItem
} // end method getPricePerItem

// return String representation of Invoice object
public String toString()
{
    return String.format( "%s: 
%s: %s (%s) 
%s: %d 
%s: $%,.2f", "invoice", "part number", getPartNumber(), getPartDescription(), "quantity", getQuantity(), "price per item", getPricePerItem() );
} // end method toString

// method required to carry out contract with interface Payable
public double getPaymentAmount()
{
    return getQuantity() * getPricePerItem(); // calculate total cost
} // end method getPaymentAmount
} // end class Invoice

Declare getPaymentAmount to fulfill contract with interface Payable
10.7.3 Creating Class Invoice

• A class can implement as many interfaces as it needs
  – Use a comma-separated list of interface names after keyword implements
    • Example: `public class ClassName extends SuperclassName implements FirstInterface, SecondInterface, ...`
public abstract class Employee implements Payable
{
    private String firstName;
    private String lastName;
    private String socialSecurityNumber;

    // three-argument constructor
    public Employee(String first, String last, String ssn)
    {
        firstName = first;
        lastName = last;
        socialSecurityNumber = ssn;
    } // end three-argument Employee constructor
}
18  // set first name
19  public void setFirstName( String first )
20  {
21       firstName = first;
22  } // end method setFirstName
23
24  // return first name
25  public String getFirstName()
26  {
27       return firstName;
28  } // end method getFirstName
29
30  // set last name
31  public void setLastName( String last )
32  {
33       lastName = last;
34  } // end method setLastName
35
36  // return last name
37  public String getLastName()
38  {
39       return lastName;
40  } // end method getLastName
```java
// set social security number
public void setSocialSecurityNumber( String ssn )
{
    socialSecurityNumber = ssn; // should validate
} // end method setSocialSecurityNumber

// return social security number
public String getSocialSecurityNumber()
{
    return socialSecurityNumber;
} // end method getSocialSecurityNumber

// return String representation of Employee object
public String toString()
{
    return String.format( "%s %s
social security number: %s",
                        getFirstName(), getLastName(), getSocialSecurityNumber() );
} // end method toString

// Note: We do not implement Payable method getPaymentAmount here so
// this class must be declared abstract to avoid a compilation error.
} // end abstract class Employee
```

getPaymentAmount method is not implemented here
10.7.5 Modifying Class SalariedEmployee for Use in the Payable Hierarchy

- Objects of any subclasses of the class that implements the interface can also be thought of as objects of the interface
  - A reference to a subclass object can be assigned to an interface variable if the superclass implements that interface
Inheritance and interfaces are similar in their implementation of the “is-a” relationship. An object of a class that implements an interface may be thought of as an object of that interface type. An object of any subclasses of a class that implements an interface also can be thought of as an object of the interface type.
// Fig. 10.14: SalariedEmployee.java
// SalariedEmployee class extends Employee, which implements Payable.

public class SalariedEmployee extends Employee
{
    private double weeklySalary;

    // four-argument constructor
    public SalariedEmployee( String first, String last, String ssn, double salary )
    {
        super( first, last, ssn ); // pass to Employee constructor
        setWeeklySalary( salary ); // validate and store salary
    } // end four-argument SalariedEmployee constructor

    // set salary
    public void setWeeklySalary( double salary )
    {
        weeklySalary = salary < 0.0 ? 0.0 : salary;
    } // end method setWeeklySalary
// return salary
public double getWeeklySalary()
{
    return weeklySalary;
} // end method getWeeklySalary

// calculate earnings; implement interface Payable method that was
// abstract in superclass Employee
public double getPaymentAmount()
{
    return getWeeklySalary();
} // end method getPaymentAmount

// return String representation of SalariedEmployee object
public String toString()
{
    return String.format("salaried employee: %s\n%s: $%,.2f", 
    super.toString(), "weekly salary", getWeeklySalary());
} // end method toString

} // end class SalariedEmployee
// Fig. 10.15: PayableInterfaceTest.java
// Tests interface Payable.

public class PayableInterfaceTest
{
    public static void main( String args[] )
    {
        // create four-element Payable array
        Payable payableObjects[ 4 ] = new Payable[ 4 ];

        // populate array with objects that implement Payable
        payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
        payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
        payableObjects[ 2 ] = new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
        payableObjects[ 3 ] = new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );

        System.out.println(  "Invoices and Employees processed polymorphically:\n" );
// generically process each element in array payableObjects
for ( Payable currentPayable : payableObjects )
{
    // output currentPayable and its appropriate payment amount
    System.out.printf( "%s 
%s: $%,.2f

", currentPayable.toString(), "payment due", currentPayable.getPaymentAmount() );
} // end for
} // end main
} // end class PayableInterfaceTest

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

Call toString and getPaymentAmount methods polymorphically
10.7.7 Declaring Constants with Interfaces

- Interfaces can be used to declare constants used in many class declarations
  - These constants are implicitly public, static and final
  - Using a static import declaration allows clients to use these constants with just their names
Software Engineering Observation 10.11

As of J2SE 5.0, it is considered a better programming practice to create sets of constants as enumerations with keyword `enum`. See Section 6.10 for an introduction to `enum` and Section 8.9 for additional `enum` details.
As you learned in Chapter 2, Java contains several comparison operators (e.g., <, <=, >, >=, ==, !=) that allow you to compare primitive values. However, these operators cannot be used to compare the contents of objects. Interface `Comparable` is used to allow objects of a class that implements the interface to be compared to one another. The interface contains one method, `compareTo`, that compares the object that calls the method to the object passed as an argument to the method. Classes must implement `compareTo` such that it returns a value indicating whether the object on which it is invoked is less than (negative integer return value), equal to (0 return value) or greater than (positive integer return value) the object passed as an argument, using any criteria specified by the programmer. For example, if class `Employee` implements `Comparable`, its `compareTo` method could compare `Employee` objects by their earnings amounts. Interface `Comparable` is commonly used for ordering objects in a collection such as an array. We use `Comparable` in Chapter 18, Generics, and Chapter 19, Collections.

A tagging interface used only to identify classes whose objects can be written to (i.e., serialized) or read from (i.e., deserialized) some type of storage (e.g., file on disk, database field) or transmitted across a network. We use `Serializable` in Chapter 14, Files and Streams, and Chapter 24, Networking.

**Fig. 10.16 | Common interfaces of the Java API.**

* (Part 1 of 2)
<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runnable</td>
<td>Implemented by any class for which objects of that class should be able to execute in parallel using a technique called multithreading (discussed in Chapter 23, Multithreading). The interface contains one method, <code>run</code>, which describes the behavior of an object when executed.</td>
</tr>
<tr>
<td>GUI event-listener interfaces</td>
<td>You work with Graphical User Interfaces (GUIs) every day. For example, in your Web browser, you might type in a text field the address of a Web site to visit, or you might click a button to return to the previous site you visited. When you type a Web site address or click a button in the Web browser, the browser must respond to your interaction and perform the desired task for you. Your interaction is known as an event, and the code that the browser uses to respond to an event is known as an event handler. In Chapter 11, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2, you will learn how to build Java GUIs and how to build event handlers to respond to user interactions. The event handlers are declared in classes that implement an appropriate event-listener interface. Each event listener interface specifies one or more methods that must be implemented to respond to user interactions.</td>
</tr>
<tr>
<td>SwingConstants</td>
<td>Contains a set of constants used in GUI programming to position GUI elements on the screen. We explore GUI programming in Chapters 11 and 22.</td>
</tr>
</tbody>
</table>
Fig. 10.17 | MyShape hierarchy.
Fig. 10.18 | MyShape hierarchy with MyBoundedShape.
Fig. 10.19 | Attributes and operations of classes BalanceInquiry, Withdrawal, and Deposit.
10.9 (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System

• **UML model for inheritance**
  – The generalization relationship
    • The superclass is a generalization of the subclasses
    • The subclasses are specializations of the superclass

• **Transaction superclass**
  – Contains the methods and fields `BalanceInquiry`, `Withdrawal` and `Deposit` have in common
    • `execute` method
    • `accountNumber` field
Fig. 10.20 | Class diagram modeling generalization of superclass `Transaction` and subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. Note that abstract class names (e.g., `Transaction`) and method names (e.g., `execute` in class `Transaction`) appear in italics.
Fig. 10.21 | Class diagram of the ATM system (incorporating inheritance). Note that abstract class names (e.g., Transaction) appear in italics.
A complete class diagram shows all the associations among classes and all the attributes and operations for each class. When the number of class attributes, methods and associations is substantial (as in Fig. 10.21 and Fig. 10.22), a good practice that promotes readability is to divide this information between two class diagrams—one focusing on associations and the other on attributes and methods.
10.9 (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System (Cont.)

- Incorporating inheritance into the ATM system design
  - If class A is a generalization of class B, then class B extends class A
  - If class A is an abstract class and class B is a subclass of class A, then class B must implement the abstract methods of class A if class B is to be a concrete class
Fig. 10.22 | Class diagram with attributes and operations (incorporating inheritance). Note that abstract class names (e.g., Transaction) and method names (e.g., execute in class Transaction) appear in italic.
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal extends Transaction
{
} // end class Withdrawal

Subclass Withdrawal extends superclass Transaction
public class Withdrawal extends Transaction {

    // attributes
    private double amount; // amount to withdraw
    private Keypad keypad; // reference to keypad
    private CashDispenser cashDispenser; // reference to cash dispenser

    // no-argument constructor
    public Withdrawal() {
    } // end no-argument Withdrawal constructor

    // method overriding execute
    public void execute() {
    } // end method execute

} // end class Withdrawal
Several UML modeling tools convert UML-based designs into Java code and can speed the implementation process considerably. For more information on these tools, refer to the Internet and Web Resources listed at the end of Section 2.9.
// Abstract class Transaction represents an ATM transaction
public abstract class Transaction
{
    // attributes
    private int accountNumber; // indicates account involved
    private Screen screen; // ATM's screen
    private BankDatabase bankDatabase; // account info database

    // no-argument constructor invoked by subclasses using super()
    public Transaction()
    {
    } // end no-argument Transaction constructor

    // return account number
    public int getAccountNumber()
    {
    } // end method getAccountNumber
// return reference to screen
public Screen getScreen()
{
} // end method getScreen

// return reference to bank database
public BankDatabase getBankDatabase()
{
} // end method getBankDatabase

// abstract method overridden by subclasses
public abstract void execute();
} // end class Transaction

Declare **abstract** method **execute**