6b

Object-Oriented Programming: Polymorphism

OBJECTIVES

In this lecture you will learn:

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

6b.1 Introduction
6b.2 Polymorphism Examples
6b.3 Demonstrating Polymorphic Behavior
6b.4 Abstract Classes and Methods
6b.5 Case Study: Payroll System Using Polymorphism
6b.5.1 Creating Abstract Superclass `Employee`
6b.5.2 Creating Concrete Subclass `SalariedEmployee`
6b.5.3 Creating Concrete Subclass `HourlyEmployee`
6b.5.4 Creating Concrete Subclass `CommissionEmployee`
6b.5.5 Creating Indirect Concrete Subclass `BasePlusCommissionEmployee`
6b.5.6 Demonstrating Polymorphic Processing, Operator `instanceof` and Downcasting
6b.5.7 Summary of the Allowed Assignments Between Superclass and Subclass Variables
6b.6 `final` Methods and Classes

6b.7 Case Study: Creating and Using Interfaces

- 6b.7.1 Developing a `Payable` Hierarchy
- 6b.7.2 Declaring Interface `Payable`
- 6b.7.3 Creating Class `Invoice`
- 6b.7.4 Modifying Class `Employee` to Implement Interface `Payable`
- 6b.7.5 Modifying Class `SalariedEmployee` for Use in the `Payable` Hierarchy
- 6b.7.6 Using Interface `Payable` to Process `Invoices` and `Employees` Polymorphically
- 6b.7.7 Declaring Constants with Interfaces
- 6b.7.8 Common Interfaces of the Java API

6b.8 (Optional) GUI and Graphics Case Study: Drawing with Polymorphism

6b.9 (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System

6b.10 Wrap-Up

6b.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**

6b.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**

Software Engineering Observation 6b.1

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

Software Engineering Observation 6b.2

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

6b.3 Demonstrating Polymorphic Behavior

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

Outline

PolymorphismTest.java

(1 of 2)

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

Typical reference assignments

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

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

PolymorphismTest.java

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

```
Call CommissionEmployee3's toString with supercla
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
```

6b.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**

6b.4 Abstract Classes and Methods (Cont.)

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

Software Engineering Observation 6b.3

An abstract class declares common attributes and behaviors of the various classes in a class hierarchy. An abstract class typically contains one or more abstract methods that subclasses must override if the subclasses 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 6b.1

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

Common Programming Error 6b.2

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

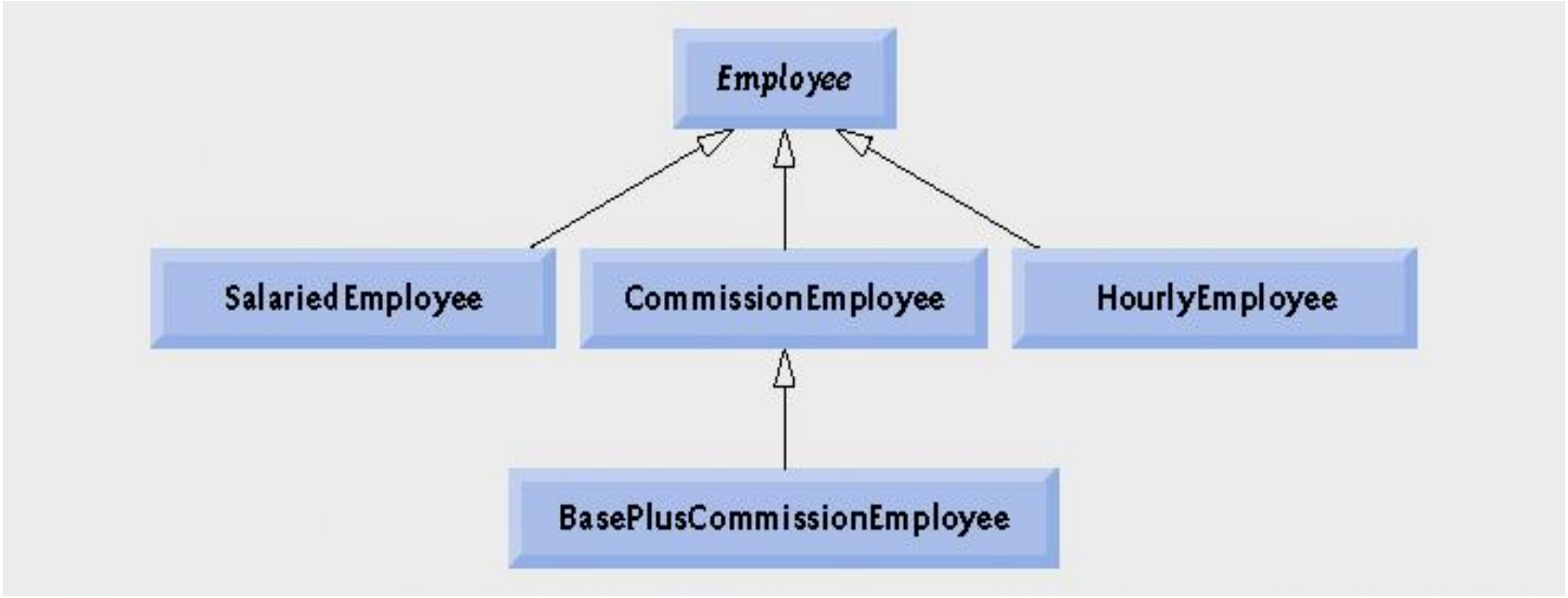

Fig. 6b.2 | `Employee` hierarchy UML class diagram.

Software Engineering Observation 6b.4

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

Software Engineering Observation 6b.4

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

6b.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**

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

Fig. 6b.3 | Polymorphic interface for the `Employee` hierarchy classes.

Outline

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

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

Attributes common to all employees

`Employee.java`

(1 of 3)

Outline

Employee.java

(2 of 3)

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

Outline

Employee.java

(3 of 3)

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

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

Outline

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

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

Call superclass constructor

Call **setWeeklySalary** method

Validate and set weekly salary value

SalariedEmployee

.java

Outline

SalariedEmployee.java

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

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

(2 of 2)

Override **toString** method

Call superclass's version of **toString**

Outline

HourlyEmployee.java

(1 of 2)

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

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

Call superclass constructor

Validate and set hourly wage value

Outline

HourlyEmployee .java

(2 of 2)

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

Validate and set hours worked value

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

Override **toString** method

Call superclass's **toString** method

Outline

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

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

CommissionEmployee.java

(1 of 3)

Call superclass constructor

Validate and set commission rate value

Outline

CommissionEmployee.java

(2 of 3)

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

Validate and set the gross sales value

Outline

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

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

`CommissionEmployee.java`

Override **`toString`** method

(3 of 3)

Call superclass's **`toString`** method

Outline

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

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

BasePlusCommissionEmployee.java

Call superclass constructor

(1 of 2)

Validate and set base salary value

Outline

BasePlusCommissionEmployee.java

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

Override **earnings** method

Call superclass's **earnings** method

(2 of 2)

Override **toString** method

Call superclass's **toString** method

Outline

PayrollSystemTest.java

(1 of 5)

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

Outline

PayrollSystemTest.java

(2 of 5)

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

Assigning subclass objects to supercalss variables

Implicitly and polymorphically call **`toString`**

Outline

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

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

PayrollSystemTest

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

(3 of 5)

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

Polymorphically call **earnings** method

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

```
Employees processed individually:

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

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

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

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

Outline

PayrollSystemTest.java

(4 of 5)

Outline

`PayrollSystemTest.java`

(5 of 5)

```
Employees processed polymorphically:

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

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

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

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

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

Same results as when the employees were processed individually

Base salary is increased by 10%

Each employee's type is displayed

6b.5.6 Demonstrating Polymorphic Processing, Operator `instanceof` and Downcasting

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

Common Programming Error 6b.3

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

Software Engineering Observation 6b.5

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

Common Programming Error 6b.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. An object can be cast only to its own type or to the type of one of its superclasses.

6b.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**

6b.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**

6b.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`**

Performance Tip 6b.1

The compiler can decide to inline a `final` method call and will do so for small, simple `final` methods. Inlining does not violate encapsulation or information hiding, but does improve performance because it eliminates the overhead of making a method call.

Common Programming Error 6b.5

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

Software Engineering Observation 6b.6

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.

6b.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**

Good Programming Practice 6b.1

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

Common Programming Error 6b.6

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

6b.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 methods of an interface**

Good Programming Practice 6b.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 a broad range of unrelated classes.

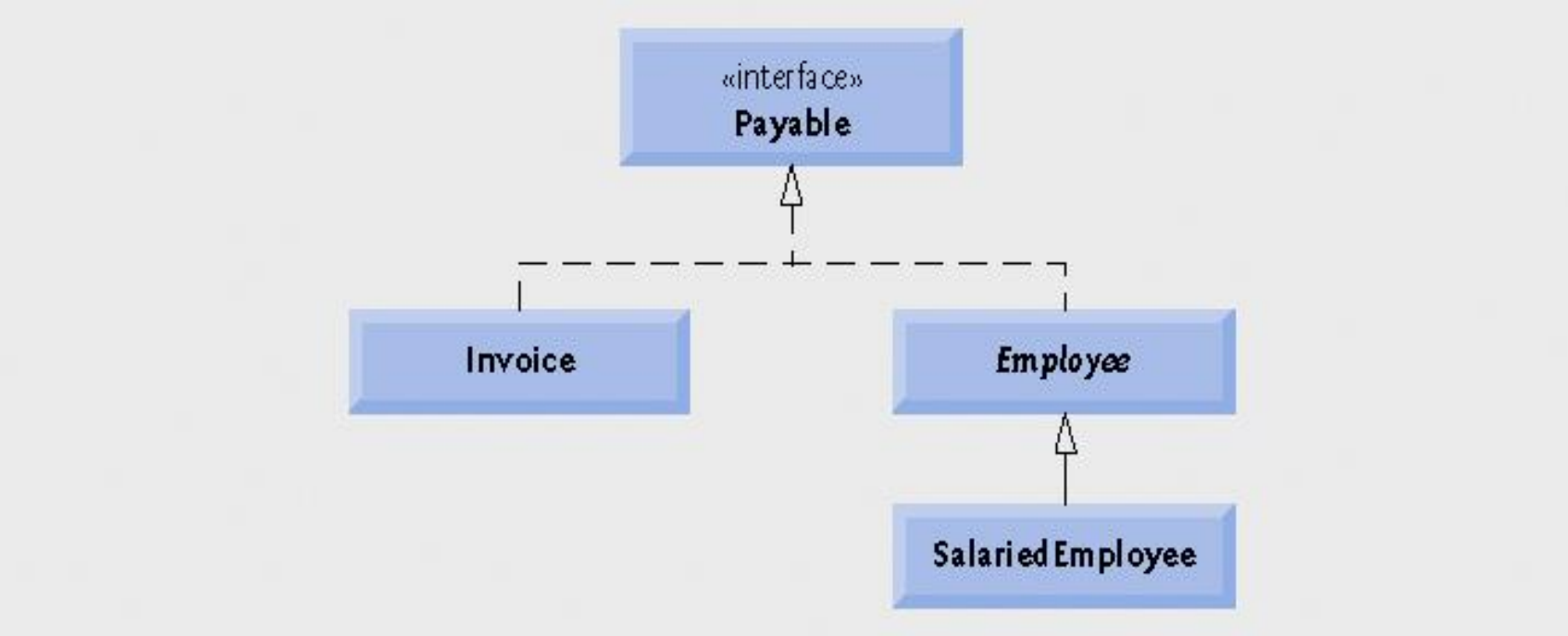

Fig. 6b.10 | Payable interface hierarchy UML class diagram.

Outline

Payable.java

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

Declare interface **Payable**

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

Outline

Invoice.java

(1 of 3)

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

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

Outline

Invoice.java

(2 of 3)

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

Outline

Invoice.java

(3 of 3)

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

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

6b.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*`, ...`**

Outline

Employee.java

(1 of 3)

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

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

Outline

Employee.java

(2 of 3)

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

Outline

Employee.java

(3 of 3)

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

`getPaymentAmount` method is not implemented here

6b.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**

Software Engineering Observation 6b.7

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

Outline

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

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

SalariedEmployee

.java

(1 of 2)

Outline

SalariedEmployee.java

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

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

Software Engineering Observation 6b.8

The “is-a” relationship that exists between superclasses and subclasses, and between interfaces and the classes that implement them, holds when passing an object to a method. When a method parameter receives a variable of a superclass or interface type, the method processes the object received as an argument polymorphically.

Software Engineering Observation 6b.9

Using a superclass reference, we can polymorphically invoke any method specified in the superclass declaration (and in class `Object`). Using an interface reference, we can polymorphically invoke any method specified in the interface declaration (and in class `Object`).

Outline

PayableInterface Test.java

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

Declare array of **Payable** variables

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

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

Outline

PayableInterface

Test.java

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

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

(2 of 2)

```
Invoices and Employees processed polymorphically:

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

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

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

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

Software Engineering Observation 6b.10

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

6b.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 6b.11

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

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

Fig. 6b.16 | Common interfaces of the Java API.
(Part 1 of 2)

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

Fig. 6b.16 | Common interfaces of the Java API. (Part 2 of 2)

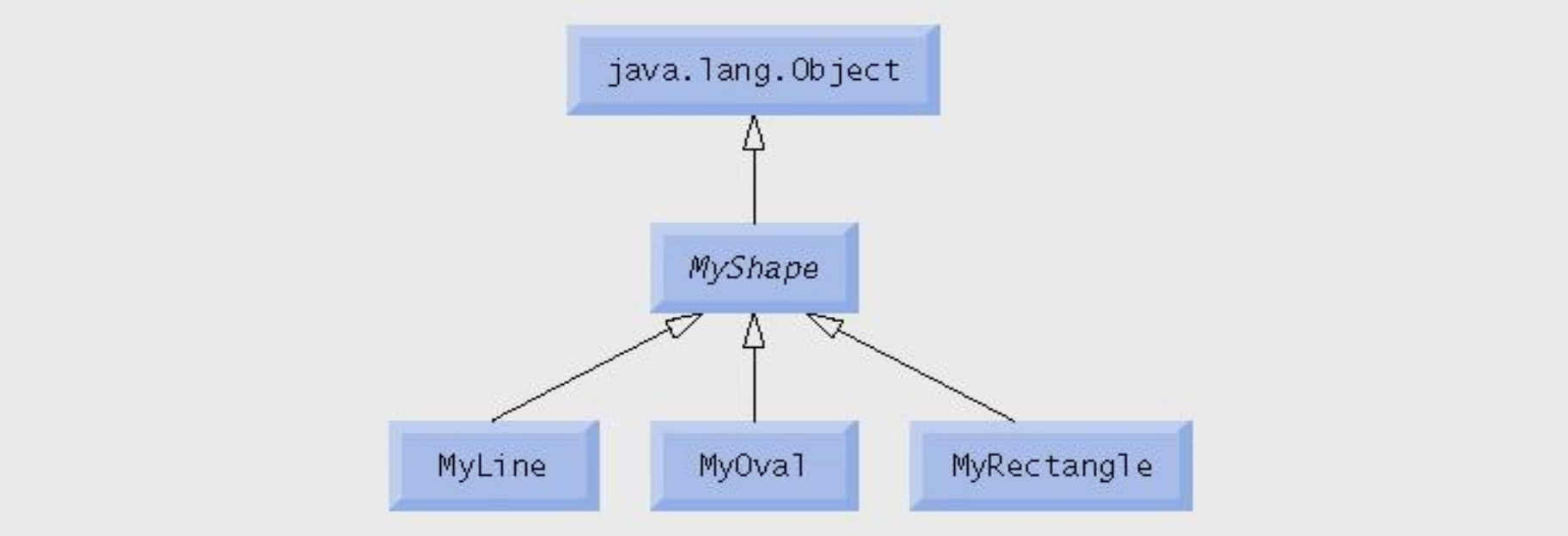

Fig. 6b.17 | MyShape hierarchy.

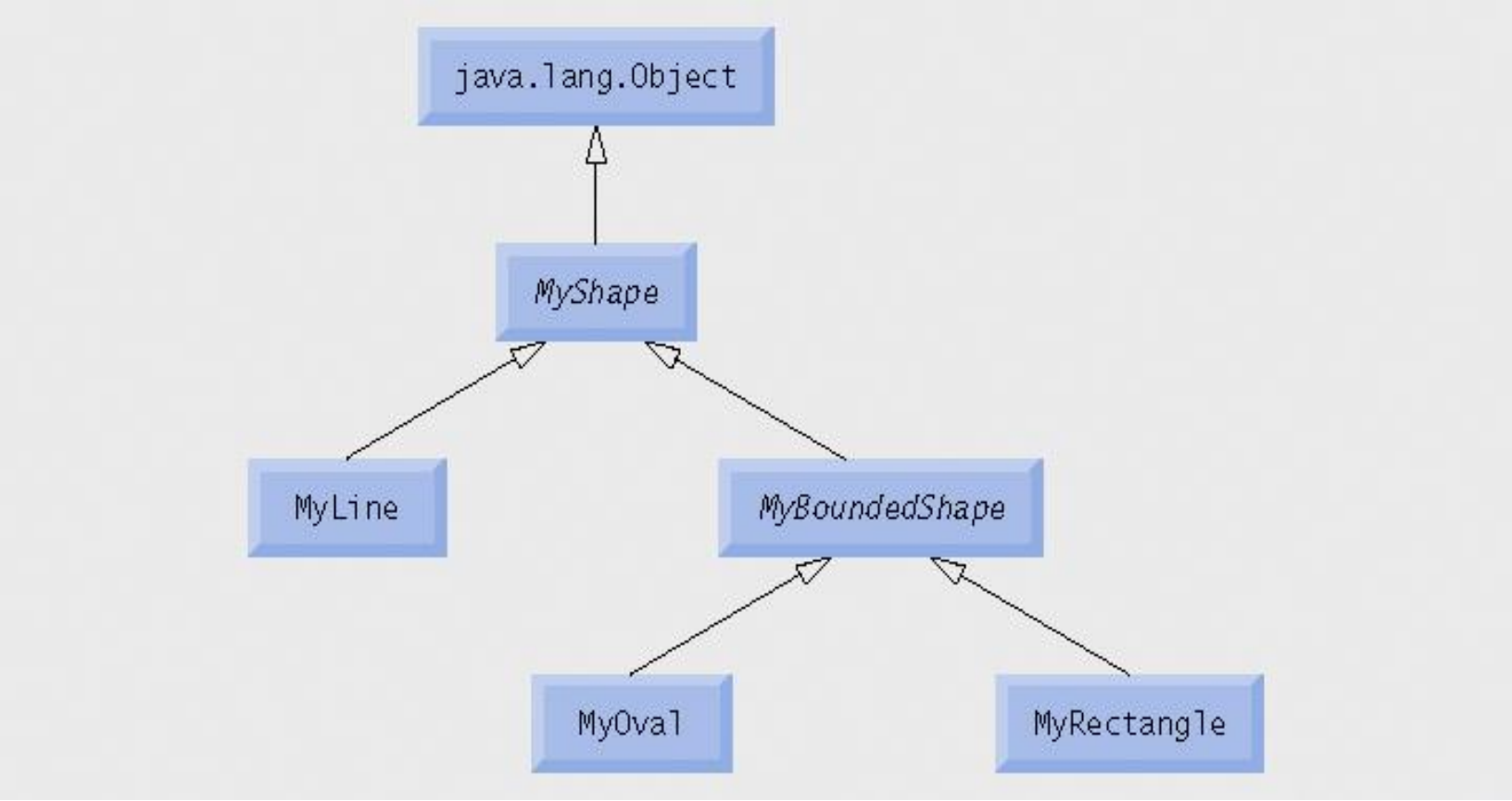

Fig. 6b.18 | MyShape hierarchy with MyBoundedShape.

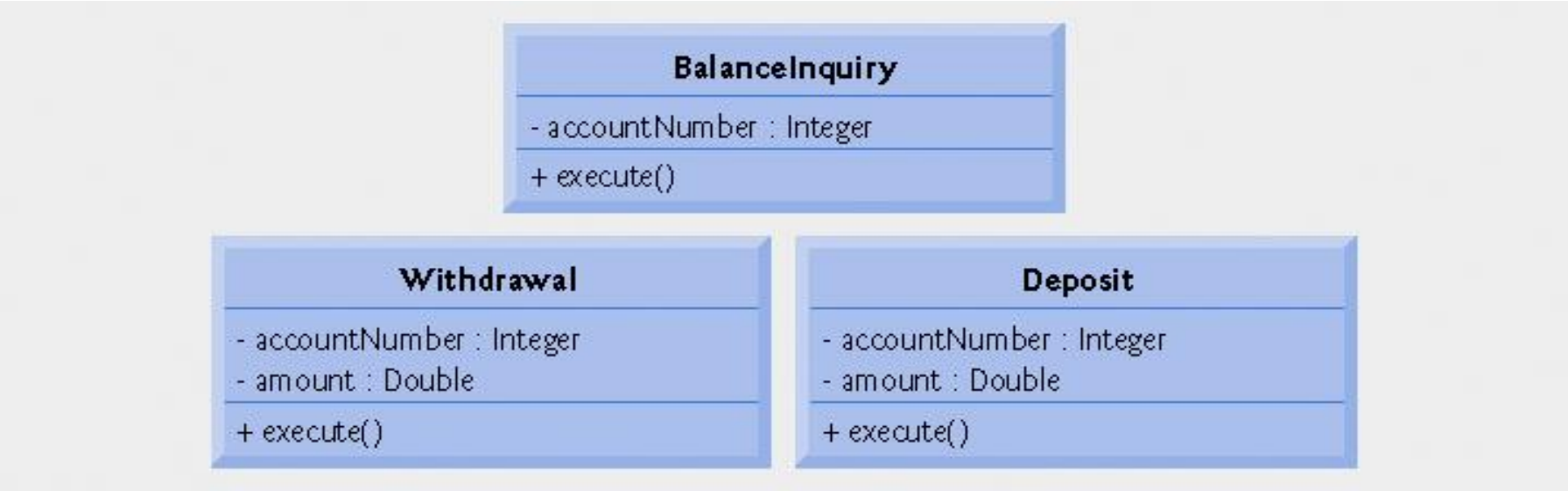

Fig. 6b.19 | Attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.

6b.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**

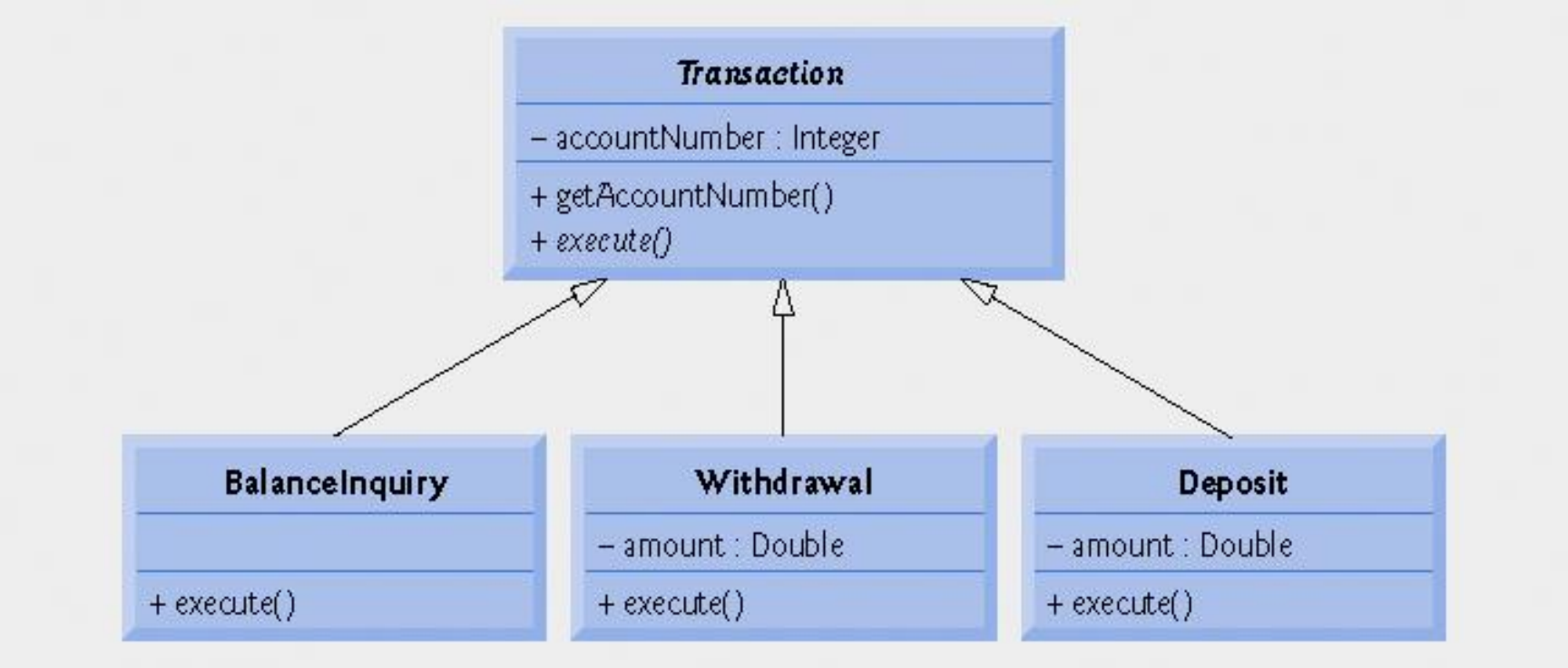

Fig. 6b. 20 | Class diagram modeling generalization of superclass `Transaction` and subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. Note that abstract class names (e.g., `Transaction`) and method names (e.g., `execute` in class `Transaction`) appear in italics.

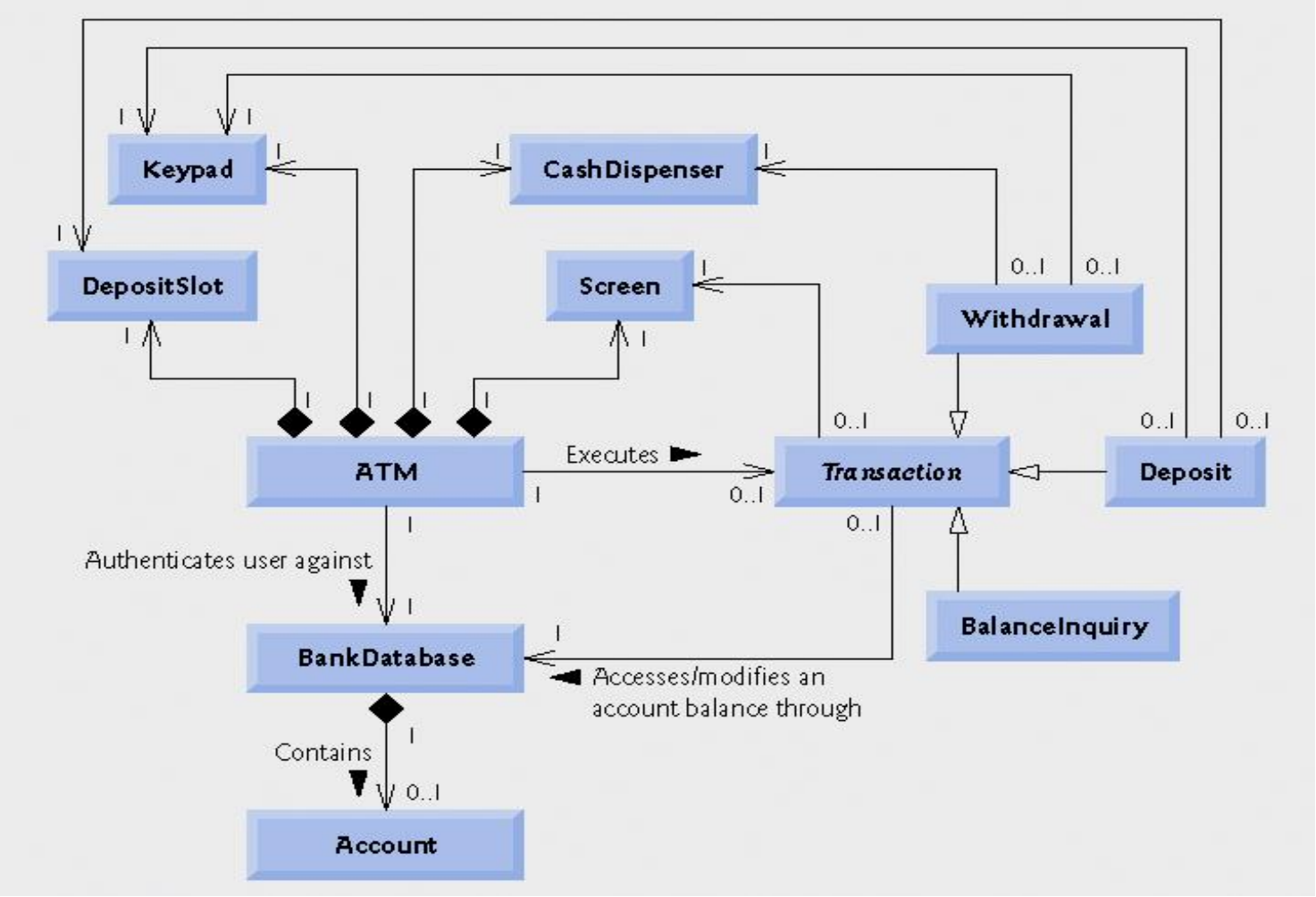

Fig. 6b.21 | Class diagram of the ATM system (incorporating inheritance). Note that abstract class names (e.g., `Transaction`) appear in italics.

Software Engineering Observation 6b.12

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. 6b.21 and Fig. 6b.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.

6b.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**

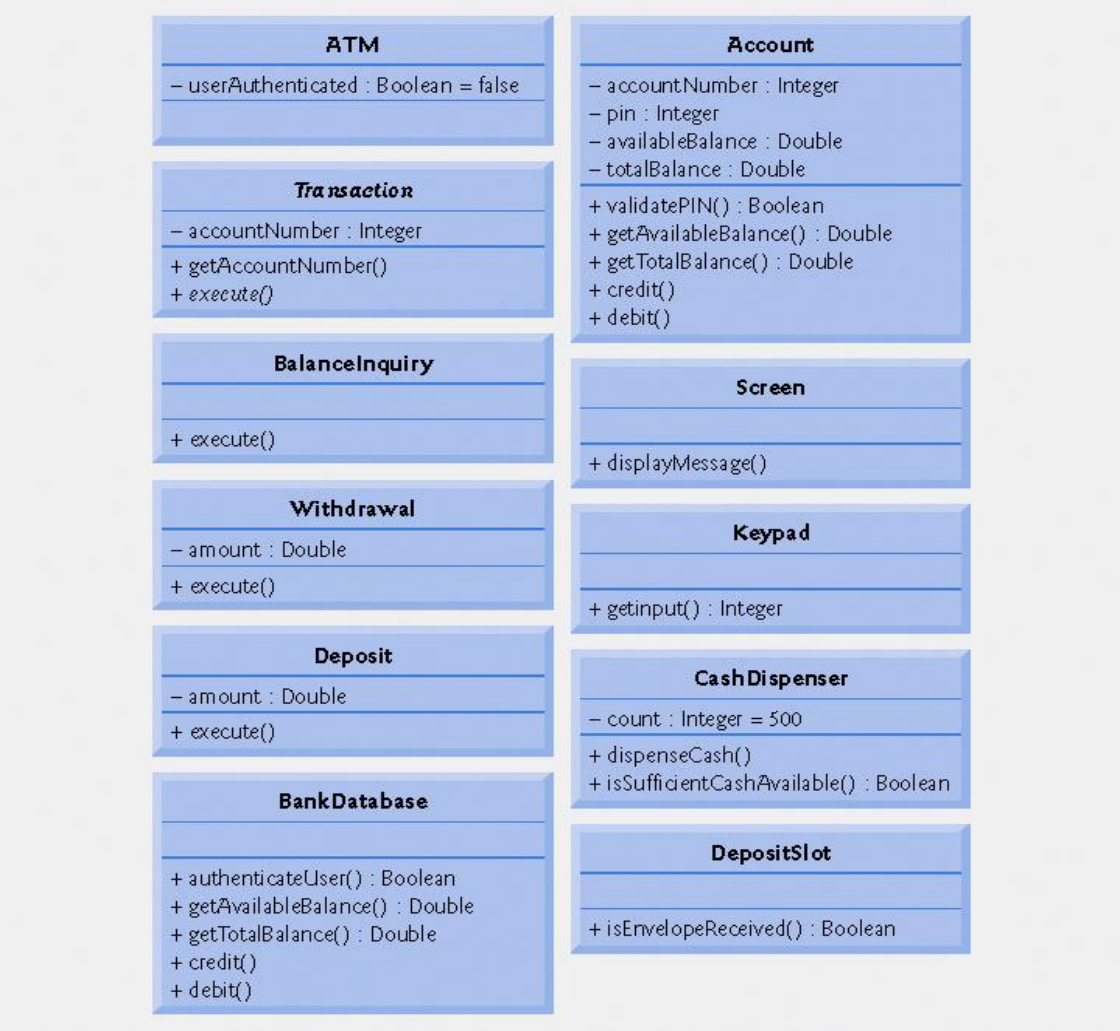

Fig. 6b.22 | Class diagram with attributes and operations (incorporating inheritance). Note that abstract class names (e.g., `Transaction`) and method names (e.g., `execute` in class `Transaction`) appear in italic

Outline

```
1 // Class Withdrawal represents an ATM withdrawal transaction
2 public class Withdrawal extends Transaction
3 {
4 } // end class Withdrawal
```

Subclass **`Withdrawal`** extends superclass **`Transaction`**

`Withdrawal.java`

Outline

```
1  // Withdrawal.java
2  // Generated using the class diagrams in Fig. 10.21 and Fig. 10.22
3  public class Withdrawal extends Transaction
4  {
5     // attributes
6     private double amount; // amount to withdraw
7     private Keypad keypad; // reference to keypad
8     private CashDispenser cashDispenser; // reference to cash dispenser
9
10    // no-argument constructor
11    public Withdrawal()
12    {
13    } // end no-argument Withdrawal constructor
14
15    // method overriding execute
16    public void execute()
17    {
18    } // end method execute
19 } // end class Withdrawal
```

Subclass **`Withdrawal`** extends superclass **`Transaction`**

`Withdrawal.java`

Software Engineering Observation 6b.13

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.

Outline

```
1  // Abstract class Transaction represents an ATM transaction
2  public abstract class Transaction
3  {
4     // attributes
5     private int accountNumber; // indicates account involved
6     private Screen screen; // ATM's screen
7     private BankDatabase bankDatabase; // account info database
8
9     // no-argument constructor invoked by subclasses using super()
10    public Transaction()
11    {
12    } // end no-argument Transaction constructor
13
14    // return account number
15    public int getAccountNumber()
16    {
17    } // end method getAccountNumber
18
```

Declare **`abstract`** superclass **`Transaction`**

Transaction.java

(1 of 2)

Outline

Transaction.java

(2 of 2)

```
19    // return reference to screen
20    public Screen getScreen()
21    {
22    } // end method getScreen
23
24    // return reference to bank database
25    public BankDatabase getBankDatabase()
26    {
27    } // end method getBankDatabase
28
29    // abstract method overridden by subclasses
30    public abstract void execute();
31 } // end class Transaction
```

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