Лекция 10

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.**

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 Definition

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**

Definition: "Content determines Behavior"

Software Engineering Observation

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

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.

10.3 Demonstrating Polymorphic Behavior

The Content of a superclass reference can be 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(Content) , not the type of the reference, determines which method is called (Behavior)**

A subclass reference can be aimed at a superclass object only if the object is downcasted.

Outline

```
// 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
      BasePlusCommissionEmployee4 basePlusCommissionEmployee =
         new BasePlusCommissionEmployee4(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // invoke toString on superclass object using superclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "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:\n\n%s\n\n",
         "Call BasePlusCommissionEmployee4's toString with subclass",
         "reference to subclass object",
         basePlusCommissionEmployee.toString() );

```

Typical reference assignments

PolymorphismTest

.java

(1 of 2)

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

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

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

PolymorphismTest .java

```
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
```

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

Software Engineering Observation

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

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

Common Programming Error

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

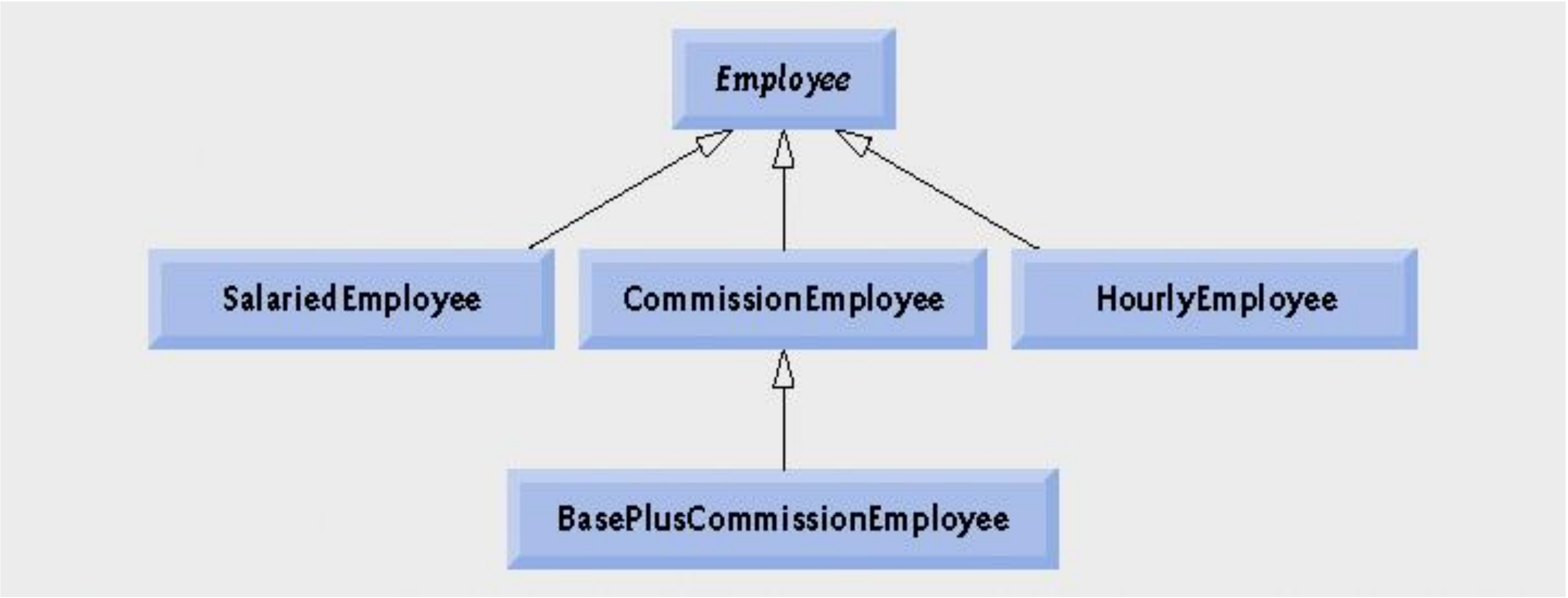


Fig. 10.2 | `Employee` hierarchy UML class diagram.

Software Engineering Observation

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

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.

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

10.5.1 Creating Abstract Superclass `Employee`

Review UML notations in class diagrams

+	Public
-	Private
#	Protected
~	Package (default visibility)

underline	static
italic	abstract
all-caps	constants

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

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

Outline

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

Attributes common to all employees

Employee.java

(1 of 3)

```java
// Fig. 10.4: Employee.java
// Employee abstract superclass.

public abstract class Employee
{
   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

```

Outline

```
   // 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

```

Employee.java

```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\nsocial 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

Employee.java

(3 of 3)

```java
// 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

```

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

Call superclass constructor

Call **setWeeklySalary** method

Validate and set weekly salary value

Outline

SalariedEmployee.java

(2 of 2)

```
    // 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\n%s: $%,.2f",
          super.toString(), "weekly salary", getWeeklySalary() );
    } // end method toString
 } // end class SalariedEmployee
```

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

Override **toString** method

Call superclass's version of **toString**

Outline

```java
// Fig. 10.6: HourlyEmployee.java
// HourlyEmployee class extends Employee.

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

```

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

Call superclass constructor

Validate and set hourly wage value

HourlyEmployee

.java

```java
   // 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\n%s: $%,.2f; %s: %,.2f",
         super.toString(), "hourly wage", getWage(),
         "hours worked", getHours() );
   } // end method toString
} // 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

```
// Fig. 10.7: CommissionEmployee.java
// CommissionEmployee class extends Employee.

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

```

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

Call superclass constructor

Validate and set commission rate value

`CommissionEmployee.java`

(1 of 3)

Outline

```
   // 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

```

Validate and set the gross sales value

CommissionEmployee.java

(2 of 3)

```java
   // 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

BasePlusCommissionEmployee.java


```
// Fig. 10.8: BasePlusCommissionEm
// BasePlusCommissionEmployee clas

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

```

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

Call superclass constructor

Validate and set base salary value

```java
   // 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
```

Override **earnings** method

Call superclass's **earnings** method

Override **toString** method

Call superclass's **toString** method

```java
// 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" );

```

Outline

PayrollSystemTest

.java


```
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         salariedEmployee, "earned", salariedEmployee.earnings() );
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         hourlyEmployee, "earned", hourlyEmployee.earnings() );
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         commissionEmployee, "earned", commissionEmployee.earnings() );
      System.out.printf( "%s\n%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;
      employees[ 3 ] = basePlusCommissionEmployee;

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

      // generically process each element in array employees
      for ( Employee currentEmployee : employees )
      {
         System.out.println( currentEmployee ); // invokes toString

```

Assigning subclass objects to supercalss variables

Implicitly and polymorphically call **toString**

```
            // determine whether element is a BasePlusCommissionEmployee
            if ( currentEmployee instanceof BasePlusCommissionEmployee )
            {
               // downcast Employee reference to
               // BasePlusCommissionEmployee reference
               BasePlusCommissionEmployee employee =
                  ( BasePlusCommissionEmployee ) currentEmployee;

               double oldBaseSalary = employee.getBaseSalary();
               employee.setBaseSalary( 1.10 * oldBaseSalary );
               System.out.printf(
                  "new base salary with 10%% increase is: $%,.2f\n",
                  employee.getBaseSalary() );
            } // end if

            System.out.printf(
               "earned $%,.2f\n\n", currentEmployee.earnings() );
         } // end for

         // get type name of each object in employees array
         for ( int j = 0; j < employees.length; j++ )
            System.out.printf( "Employee %d is a %s\n", j,
               employees[ j ].getClass().getName() );
      } // end main
} // end class PayrollSystemTest
```

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

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

Give **BasePlusCommissionEmployees** 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

Base salary is increased by 10%

Each employee's type is displayed

10.5.6 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

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

Software Engineering Observation

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

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.

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`

Performance Tip

The compiler can decide to inline a `final` method call and will do so for small, simple `final` methods. It is good practice to declare Setter and Getter methods `final`—typically for security reasons.

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

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

Software Engineering Observation

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 Enumerations

```
// int Enum Pattern - has severe problems!
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL   = 3;
```

This pattern has many problems, such as:

- **Not type-safe - Since a season is just an `int` you can pass in any other int value where a season is required, or add two seasons together (which makes no sense).**
- **No namespace - You must prefix constants of an `int` enum with a string (in this case SEASON_) to avoid collisions with other `int` enum types.**
- **Brittleness - Because `int` enums are compile-time constants, they are compiled into clients that use them. If a new constant is added between two existing constants or the order is changed, clients must be recompiled. If they are not, they will still run, but their behavior will be undefined.**
- **Printed values are uninformative - Because they are just `int`s, if you print one out all you get is a number, which tells you nothing about what it represents, or even what type it is.**

10.7 Enumerations

`enum` types- a reference type, default value is `null`

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

10.7 Enumerations

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
 }
 public enum Status{ CONTINUE, WON, LOST};
    // static constants!
    // …. in some method
    Status mode = Status.WON;
    // …change it
    mode = Status.LOST;
```

10.7 Enumerations

```
enum Animals {
  DOG("woof"), CAT("meow"), FISH("burble");
  final String sound; // package access
  private Animals(String s) { sound = s; }
}
class TestEnum{
  static Animals a;// undefined
  public static void main (String[] args){
    System.out.println(a.DOG.sound + " "
                                  + a.FISH.sound
                                  + " "    +   a);
  }
}
```

```
run:
woof burble null
BUILD SUCCESSFUL (total time: 1 second)
```

10.7 Enumerations

a.DOG.sound transfroms into Animals.DOG.sound because enum vars are implicitly static. In fact enum converts to a Java class inheriting the functionality of class java.lang.Enum. Therefore, after compilation enum Animals gets the form

```
class Animals extends java.lang.Enum {
  public static final Animals DOG = new Animals("woof");
  public static final Animals CAT = new Animals("meow");
  public static final Animals FISH = new Animals("burble");
  String sound;
  Animals(String s) { sound = s; }
  //compiler generates methods like toString(),equals() etc.
}
```

10.7 Enumerations

As with any `class`, it's easy to provide methods in an `enum` type which change the `state` of an `enum` constant. Thus, the term "`enum constant`" is rather misleading. What is constant is the identity of the `enum` element, not its state. Perhaps a better term would have been "`enum element`" instead of "`enum constant`".

Constructors for an `enum` type should be declared as `private`. The compiler allows non `private` declares for constructors, but this seems misleading to the reader, since `new` can never be used with enum types.

10.7 Enumerations

```
enum Flavor
{    // mutable enum state
     CHOCOLATE(100),
     VANILLA(120),
     STRAWBERRY(80);
     void setCalories(int aCalories)
     {  //changes the state of the enum 'constant'
          fCalories = aCalories;
     }
     int getCalories() { return fCalories;  }
     private Flavor(int aCalories) { fCalories = aCalories;   }
     private int fCalories;
}
private static void exerMutableEnum()
{
    Flavor.VANILLA.setCalories(75); //change the state of the enum "constant"
    System.out.println("Calories in Vanilla: " + Flavor.VANILLA.getCalories());
    Flavor.STRAWBERRY.setCalories(100); //change the state of the enum "constant"
    System.out.println("Calories in STRAWBERRY: " + Flavor.STRAWBERRY.getCalories());
    System.out.println("Calories in Vanilla: " + Flavor.VANILLA.getCalories());
}
```

```
run:
Calories in Vanilla: 75
Calories in STRAWBERRY: 100
Calories in Vanilla: 75
BUILD SUCCESSFUL (total time: 2 seconds)
```

Outline

Book.java

(1 of 2)

```java
// Fig. 8.10: Book.java
// Declaring an enum type with constructor and explicit instance fields
// and accessors for these field

public enum Book
{
   // declare constants of enum type
   JHTP6( "Java How to Program 6e", "2005" ),
   CHTP4( "C How to Program 4e", "2004" ),
   IW3HTP3( "Internet & World Wide Web How to Program 3e", "2004" ),
   CPPHTP4( "C++ How to Program 4e", "2003" ),
   VBHTP2( "Visual Basic .NET How to Program 2e", "2002" ),
   CSHARPHTP( "C# How to Program", "2002" );

   // instance fields
   private final String title; // book title
   private final String copyrightYear; // copyright year

   // enum constructor
   private Book( String bookTitle, String year )
   {
      title = bookTitle;
      copyrightYear = year;
   } // end enum Book constructor

```

Declare six **enum** constants

Arguments to pass to the **enum** constructor

Declare instance variables

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

```java
   // accessor for field title
   public String getTitle()
   {
      return title;
   } // end method getTitle

   // accessor for field copyrightYear
   public String getCopyrightYear()
   {
      return copyrightYear;
   } // end method getCopyrightYear
} // end enum Book
```

Book.java

(2 of 2)

- if an **`enum`** is a member of a **`class`**, it's implicitly **`static`**
- **`name()`** and **`valueOf()`** simply use the text of the **`enum`** constants, while **`toString()`** may be overridden to provide any content, if desired
- for **`enum`** constants, **`equals()`** and **==** amount to the same thing, and can be used interchangeably

10.7 Enumerations (Cont.)

`static` method `values`

- Generated by the compiler for every `enum`
- Returns an array of the `enum`’s constants in the order in which they were declared

`static` method `ordinal`

- Returns the sequential number of an `enum` constant

`static` method `range` of class `EnumSet`

- Takes two parameters, the first and last `enum` constants in the desired range
- Returns an `EnumSet` containing the constants in that range, inclusive
- An enhanced `for` statement can iterate over an `EnumSet` as it can over an array

EnumTest.java

```java
// Fig. 8.11: EnumTest.java
// Testing enum type Book.
import java.util.EnumSet;

public class EnumTest
{
   public static void main( String args[] )
   {
      System.out.println( "All books:\n" );

      // print all books in enum Book
      for ( Book book : Book.values() )
         System.out.printf( "%-10s%-45s%s\n", book,
             book.getTitle(), book.getCopyrightYear() );

      System.out.println( "\nDisplay a range of enum constants:\n" );

      // print first four books
      for ( Book book : EnumSet.range( Book.JHTP6, Book.CPPHTP4 ) )
         System.out.printf( "%-10s%-45s%s\n", book,
             book.getTitle(), book.getCopyrightYear() );
   } // end main
} // end class EnumTest
```

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

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

```
All books:

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

Display a range of enum constants:

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

Common Programming Error

In an `enum` declaration, it is a syntax error to declare `enum` constants after the `enum` type's constructors, fields and methods in the `enum` declaration.

```java
public enum ConvertableEnum { // convert int to enum
    POSITIVE, NEGATIVE, EITHER, UNDEFINED;

    public static ConvertableEnum convertIntToEnum(int i) {
        return values()[i];   // values() converts enum to an array
    }

    public static ConvertableEnum convertIntToEnumWithException(int i) {
        try {
            return values()[i];
        } catch (ArrayIndexOutOfBoundsException e) {
            return UNDEFINED;
        }
    }

    public static ConvertableEnum convertIntToEnumWithOrdinal(int i) {
        for (ConvertableEnum current : values()) {
            if (current.ordinal() == i) {  // Using ordinal()!!
                return current;
            }
        }

        return UNDEFINED;
    }
}
```

values() converts enum to an array

ordinal() returns the sequential number of an enum constant

Outline

Day.java

```java
public enum Day {
    SUNDAY(1),
    MONDAY(2),
    TUESDAY(3),
    WEDNESDAY(4),
    THURSDAY(5),
    FRIDAY(6),
    SATURDAY(7);
    private final int value;
    private Day(int value) {
        this.value = value;
    }
    public int getValue() {
        return this.value;
    }
    // overrides the default definition of toString() for Enumeration
    @Override
    public String toString() {
        switch(this) {
            case FRIDAY:
                return "Friday: "    + value;
            case MONDAY:
                return "Monday: "    + value;
            case SATURDAY:
                return "Saturday: " + value;
            case SUNDAY:
                return "Sunday: "    + value;
            case THURSDAY:
                return "Thursday: " + value;
            case TUESDAY:
                return "Tuesday: "   + value;
            case WEDNESDAY:
                return "Wednesday: "+ value;
            default:
                return null;
        }
    }
}
```

In order to retrieve the value of each constant of the `enum`, you can define a `public` method inside the `enum`

An enum can override the `toString()` method, just like any other Java class

Car.java

You can define **abstract** methods inside an **enum** in Java. Each constant of the enum implements each **abstract** method independently.

```java
public enum Car {
    AUDI {
        @Override
        public int getPrice() {
            return 25000;
        }
    },
    MERCEDES {
        @Override
        public int getPrice() {
            return 30000;
        }
    },
    BMW {
        @Override
        public int getPrice() {
            return 20000;
        }
    };

    public abstract int getPrice();
}
```

Outline

EnumTest.java

The values inside an **enum** are constants and thus, you **can use them in comparisons** using the **equals()** or **compareTo()** methods. The Java Compiler automatically generates a **static** method for each **enum**, called **values()**. This method **returns an array of all constants** defined inside the **enum**.

```java
public static void main(String[] args) {
    System.out.println("FRIDAY".compareTo(Day.FRIDAY.name()) + "\n");
    //Printing all constants of an enum.
    for(Day day: Day.values())
        System.out.println(day.name());
    System.out.println();
    for(Day day: Day.values())
        System.out.println(day.name());
    System.out.println();
    //The following statements are illegal.
    //Day d = new Day();
    //Day.FRIDAY = Day.valueOf("New Value");
    Car c = Car.AUDI;
    System.out.println(c.name() + ": " + c.getPrice());

    Car c1 = Car.valueOf("MERCEDES");
    System.out.println(c1.toString());

    //The following statement throws an java.lang.IllegalArgumentException.
    //Car c2 = Car.valueOf("Bmw");
}
```

Outline

EnumTest.java

Output of **EnumTest**

`"FRIDAY".compareTo(Day.FRIDAY.name())`

```
0

SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY

Sunday: 1
Monday: 2
Tuesday: 3
Wednesday: 4
Thursday: 5
Friday: 6
Saturday: 7

AUDI: 25000
MERCEDES
BUILD SUCCESSFUL (total time: 0 seconds)
```

The names() of the values are returned in the same order as they were initially defined

The overridden **toString()** method

The default **toString()** method

10.7 Enumerations (Cont.)

```
enum Operation
{
    PLUS { double eval(double x, double y){ return x + y; } },
    MINUS{ double eval(double x, double y){ return x - y; } },
    TIMES{ double eval(double x, double y){ return x * y; } },
    DIVIDE{ double eval(double x, double y){ return x / y; } };
    // Do arithmetic op represented by this constant
    abstract double eval(double x, double y);
}
private static void exerEnumMethods(double x, double y)
{
    for (Operation op : Operation.values())
    {
        System.out.printf("%f %s %f = %f%n", x, op, y, op.eval(x, y));
    }
}
```

```
1.000000 PLUS 2.000000 = 3.000000
1.000000 MINUS 2.000000 = -1.000000
1.000000 TIMES 2.000000 = 2.000000
1.000000 DIVIDE 2.000000 = 0.500000
```

10.7 Enumerations (Cont.)

```
enum Direction
{// Enum types
    EAST(0)    { public String shout(){ return "Direction is East !!!"; } },
    WEST(180)  { public String shout(){ return "Direction is West !!!"; } },
    NORTH(90)  { public String shout(){ return "Direction is North !!!"; } },
    SOUTH(270) { public String shout(){ return "Direction is South !!!"; } };
    // Constructor
    private Direction(final int angle)  { this.angle = angle;
    }
    // Internal state
    private int angle;
    public int getAngle() { return angle;  }
    // Abstract method which need to be implemented</strong>
    public abstract String shout();
}
```

```
private static void exerEnumDirection()
{
    for (Direction dir : Direction.values())
    {
        System.out.printf("%s %d %s\n", dir, dir.getAngle(),dir.shout());
    }
}
```

```
run:
EAST 0 Direction is East !!!
WEST 180 Direction is West !!!
NORTH 90 Direction is North !!!
SOUTH 270 Direction is South !!!
BUILD SUCCESSFUL (total time: 1 second)
```

10.8 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

A proper style to declare an interface's methods is without using 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

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`.

10.8.1 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

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.

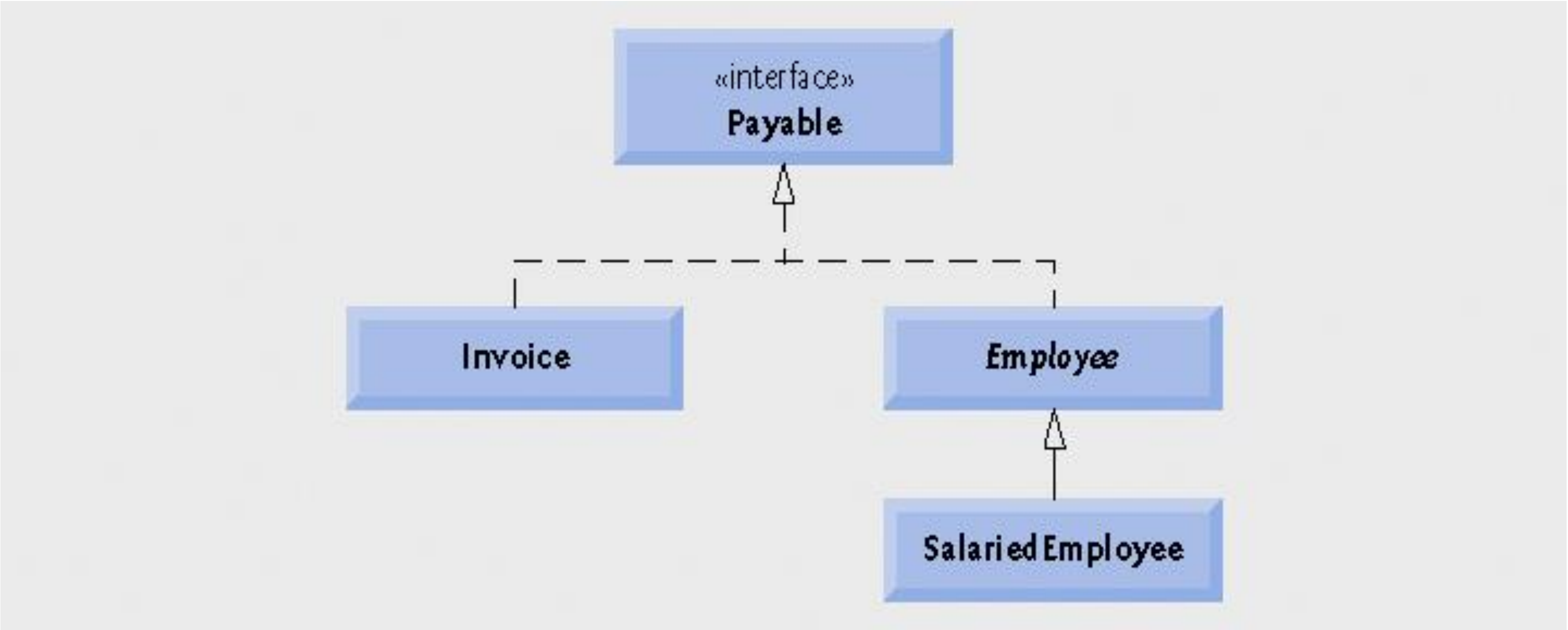


Fig. 10.10 | `Payable` interface hierarchy UML class diagram.

Outline

```
// Fig. 10.11: Payable.java
// Payable interface declaration.

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

Declare interface **Payable**

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

Payable.java

```java
// 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

```

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

```
   // 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

```

```java
   // 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: \n%s: %s (%s) \n%s: %d \n%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**

10.8.2 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* `, …`**

```java
// Fig. 10.13: Employee.java
// Employee abstract superclass implements Payable.

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

```

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

Outline

```
   // 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

```

Employee.java

```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\nsocial 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

10.8.3 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

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

```
// 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

```

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

SalariedEmployee

.java

```
   // 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
```

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

Software Engineering Observation

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

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

```java
// Fig. 10.15: PayableInterfaceTest.java
// Tests interface Payable.

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

```

Declare array of **`Payable`** variables

`PayableInterfaceTest.java`

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

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

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

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

```
Invoices and Employees processed polymorphically:

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

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

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

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

Software Engineering Observation

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`.

10.9 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

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.

10.9a Common interfaces of the Java API

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 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` with gerneric data structures.
`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` with Java Networking.

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

10.9a Common interfaces of the Java API

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. 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. 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. .

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

10.9a Common interfaces of the Java API

`Iterator` interface

- ✓ 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

`java.util` package has `public interface Iterator` and contains three methods:

- ❑ `boolean hasNext()`: It returns `true` if `Iterator` has more element to iterate.
- ❑ `Object next()`: It returns the next element in the collection until the `hasNext()`method return true. This method throws '`NoSuchElementException`' if there is no next element.
- ❑ `void remove()`: It removes the current element in the collection. This method throws '`IllegalStateException`' if this function is called before `next( )` is invoked

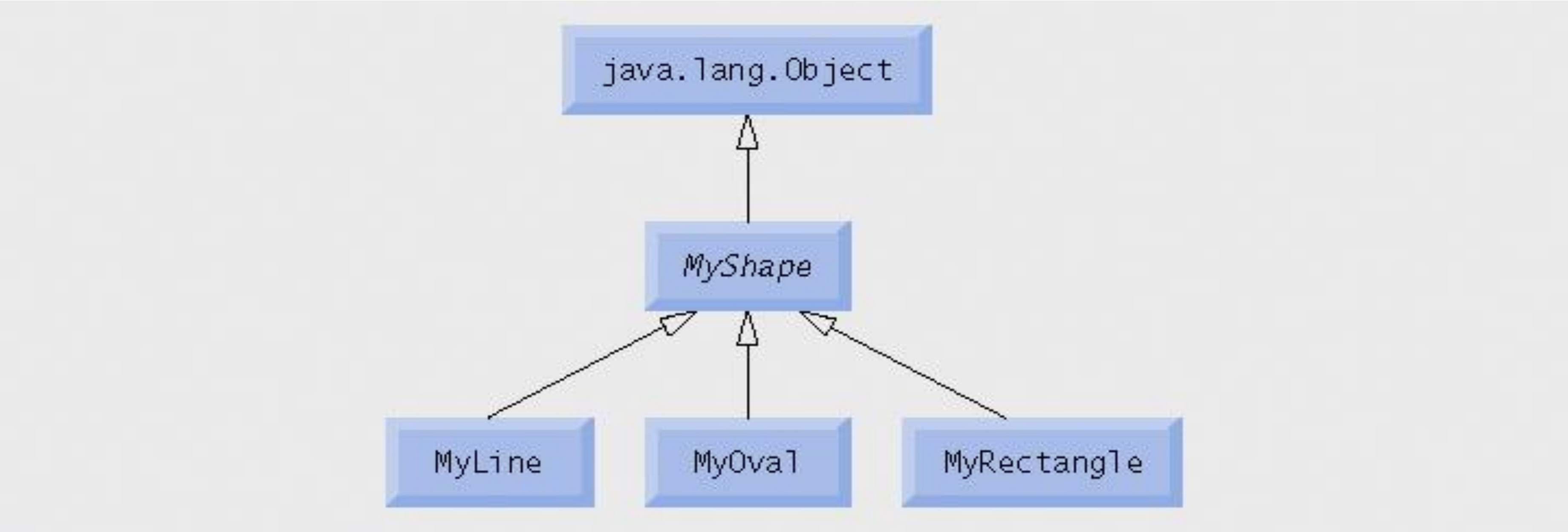


Fig. 10.17 | MyShape hierarchy.

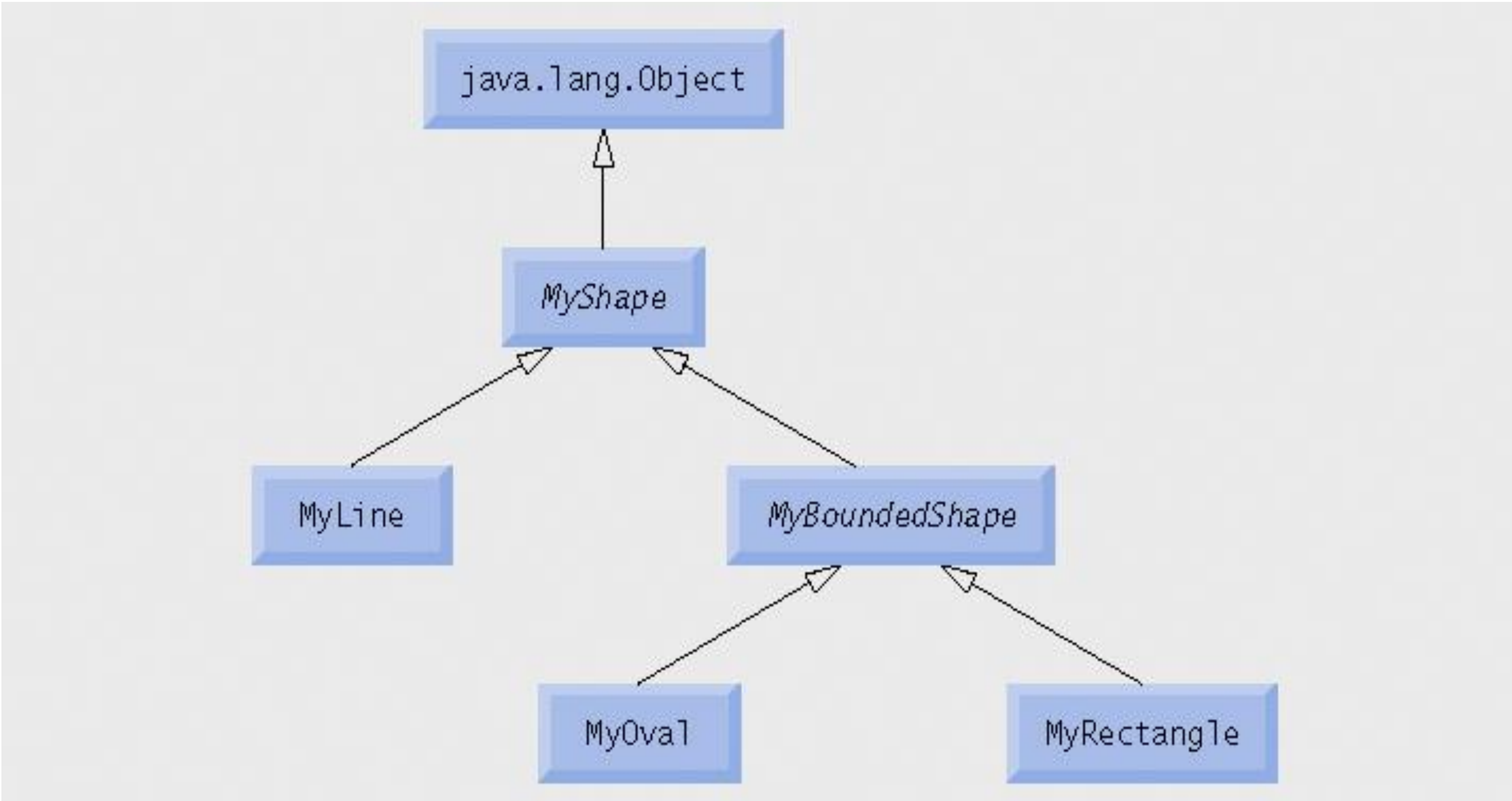


Fig. 10.18 | MyShape hierarchy with MyBoundedShape.

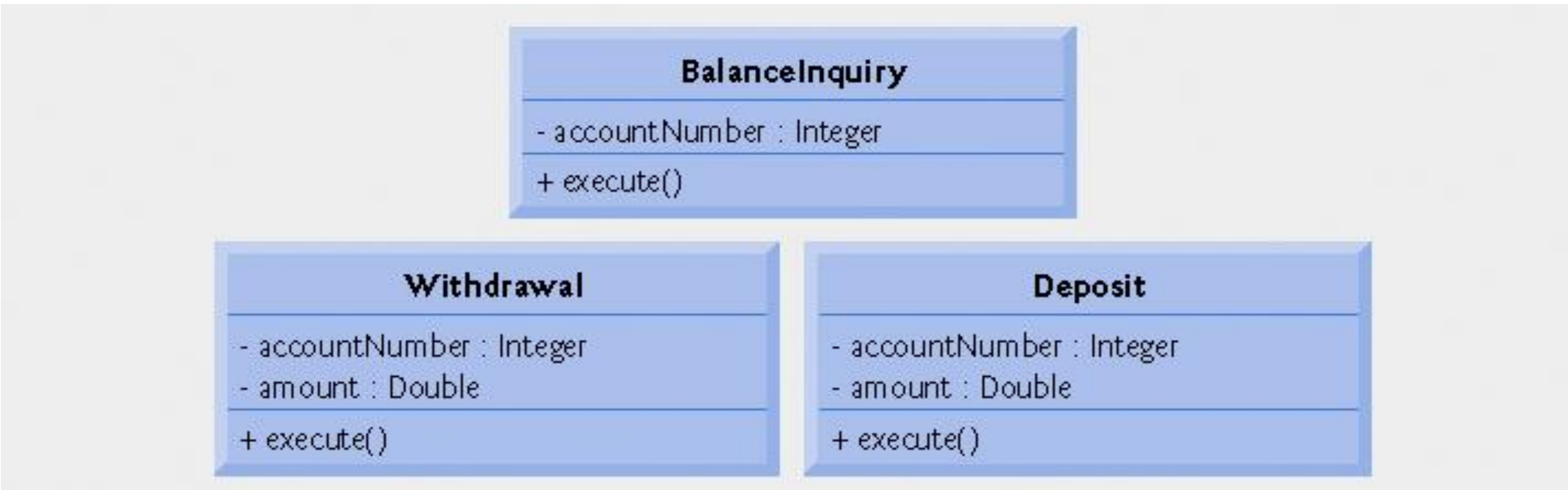
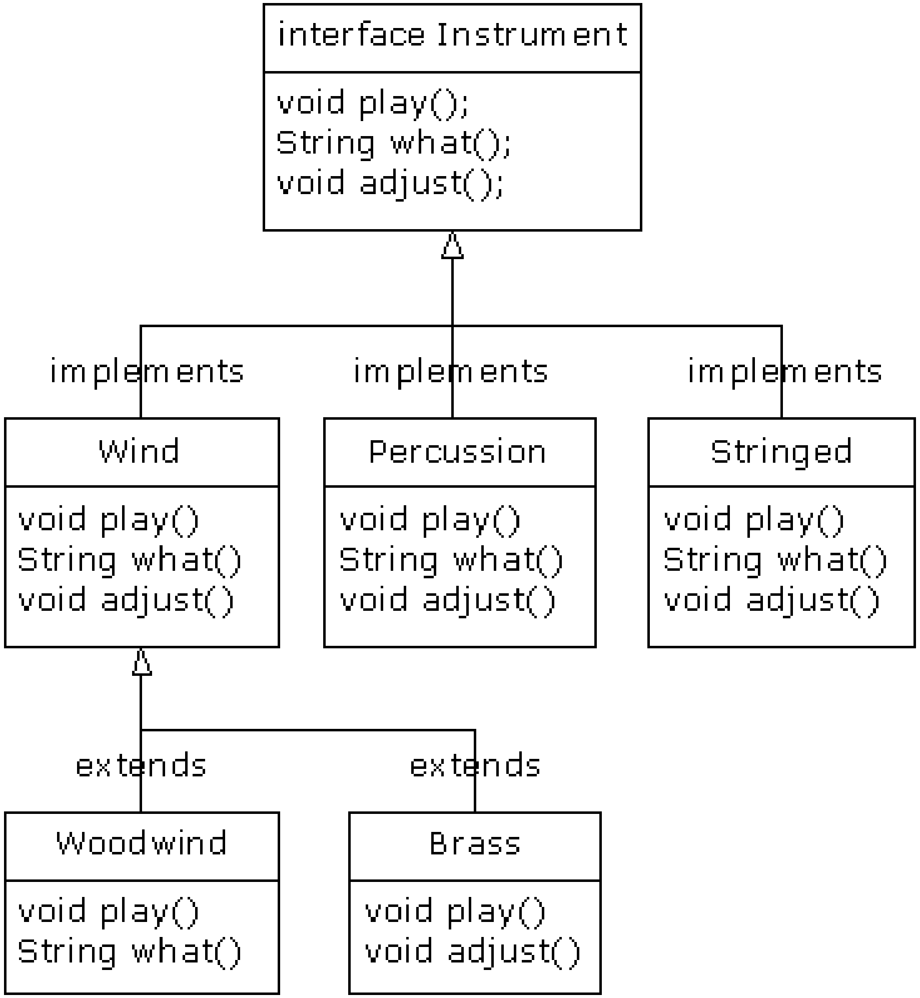


Fig. 10.19 | **Attributes and operations of classes `BalanceInquiry`, `Withdrawal` and `Deposit`.**

10.10 Software Engineering Case Study: Implementing interfaces



Резюме

The implemented methods are inherited

```
// Interfaces.
import java.util.*;

interface Instrument {
  // Compile-time constant:
  int i = 5; // static & final
  // Cannot have method definitions:
  void play(); // Automatically public
  String what();
  void adjust();
}

class Wind implements Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion implements Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed implements Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}
// continues on the next slide
```

```
class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music5 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
}
```


Method ***tuneAll*** can tune any instrument that implements ***interface Instrument***.

.

10.10 Software Engineering Case Study: Implementing interfaces

An **interface** says: “This is what all classes that *implement* this particular interface will look like.” Thus, any code that uses a particular **interface** knows what methods might be called for that **interface**, and that’s all. So the **interface** is used to establish a “protocol” between classes. (Some object-oriented programming languages have a keyword called *protocol* to do the same thing.)

10.12 Software Engineering Case Study: Implementing interfaces

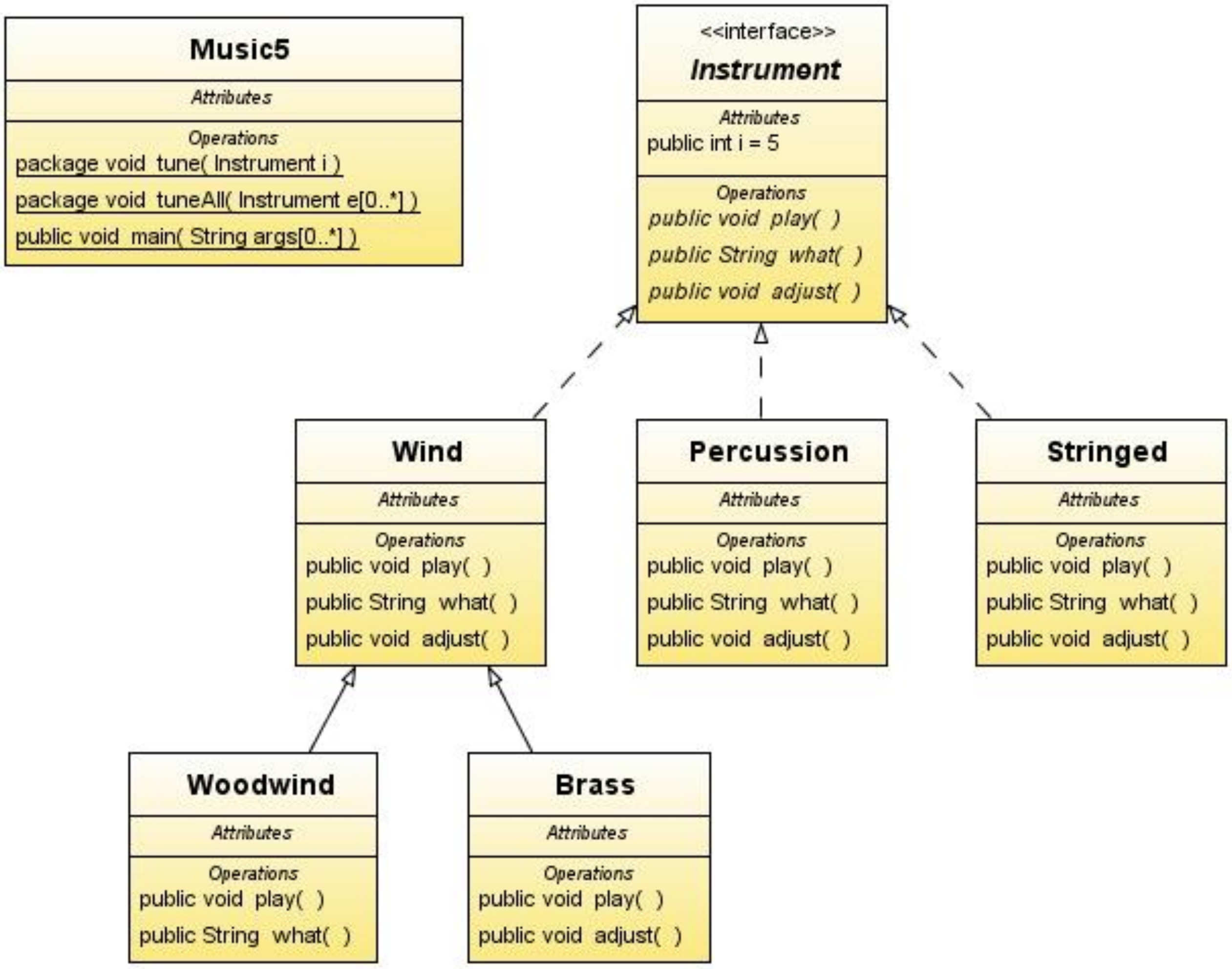
To create an **interface**, use the **interface** keyword instead of the **class** keyword. Like a class, you can add the **public** keyword before the **interface** keyword (but only if that **interface** is defined in a file of the same name) or leave it off to give package access, so that it is only usable within the same package

10.10 Software Engineering Case Study: Implementing interfaces

To make a class that conforms to a particular **interface** (or group of **interfaces**) use the **implements** keyword. **implements** says "The **interface** is what it looks like, but now I'm going to say how it *works*."

Other than that, it looks like inheritance. The UML diagram for the instrument example shows this.

UML diagram


Music5
Attributes
Operations
package void tune(Instrument i)
package void tuneAll(Instrument e[0..*])
public void main(String args[0..*])
<<interface>>
Instrument
Attributes
public int i = 5
Operations
public void play( )
public String what( )
public void adjust( )
Wind
Attributes
Operations
public void play( )
public String what( )
public void adjust( )
Percussion
Attributes
Operations
public void play( )
public String what( )
public void adjust( )
Stringed
Attributes
Operations
public void play( )
public String what( )
public void adjust( )
Woodwind
Attributes
Operations
public void play( )
public String what( )
Brass
Attributes
Operations
public void play( )
public void adjust( )

10.10.a Multiple inheritance

Java's approach to inheritance is called ***single inheritance***. This term means that a derived class can have only one parent. Some object-oriented languages allow a child class to have multiple parents. This approach is called ***multiple inheritance*** and is occasionally useful for describing objects that are in between two categories or classes.

For example, suppose we had a class **`Car`** and a class **`Truck`** and we wanted to create a new class called **`PickupTruck`**. A pickup truck is somewhat like a car and somewhat like a truck. With single inheritance, we must decide whether it is better to derive the new class from **`Car`** or **`Truck`**. With multiple inheritance, it can be derived from both.

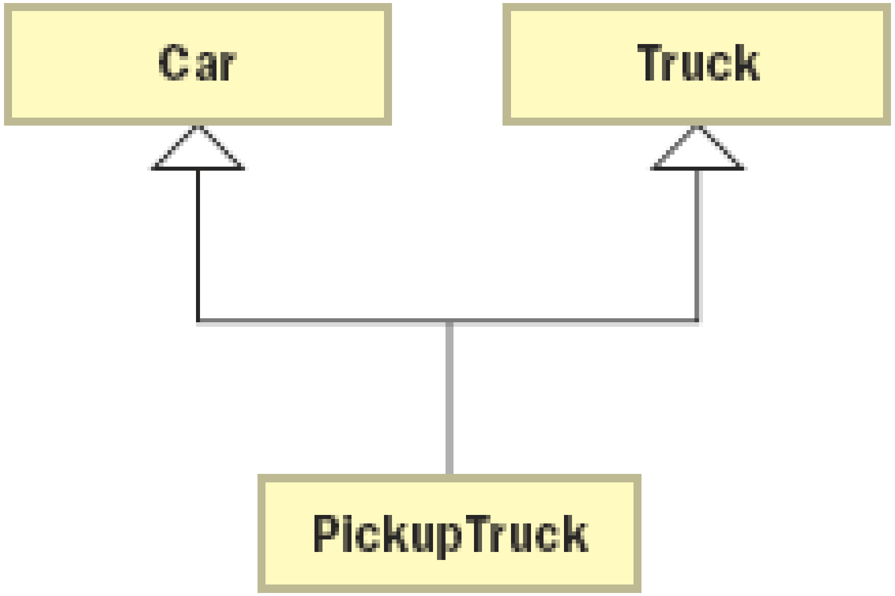


Fig. 10.23 | A UML class diagram showing multiple inheritance

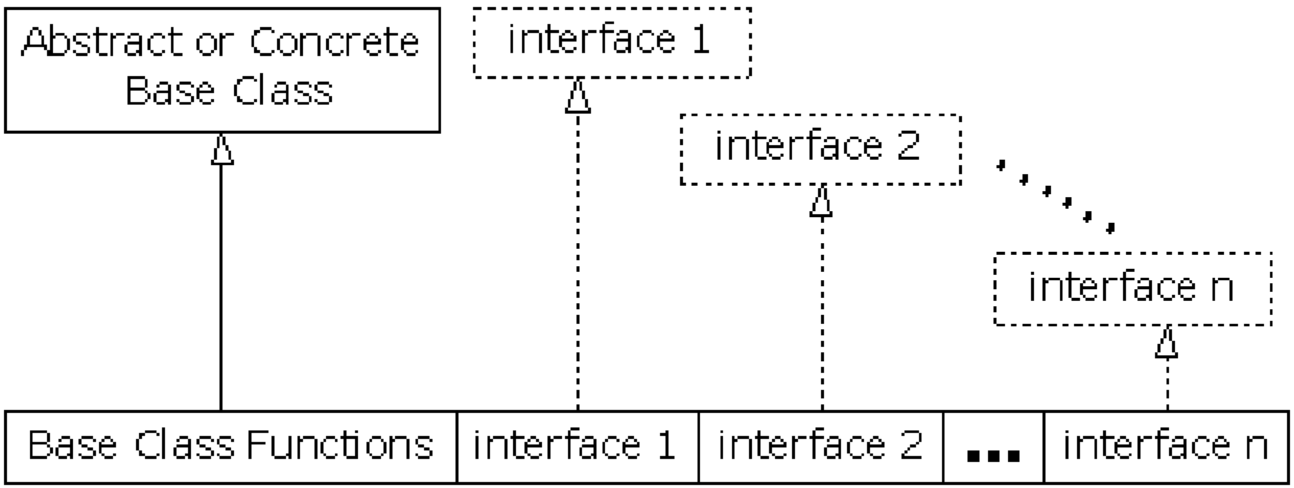
10.10.a Multiple inheritance

Multiple inheritance works well in some situations, but it comes with a price. What if both `Truck` and `Car` have methods with the same name? Which method would `PickupTruck` inherit? The answer to this question is complex, and it depends on the rules of the language that supports multiple inheritance. The designers of the Java language explicitly decided not to support multiple inheritance.

Instead, **we can rely on interfaces to provide the best features of multiple inheritance without the added complexity**. Although a Java class can be derived from only one parent class, it **can implement multiple interfaces**.

Therefore, we can interact with a particular class in specific ways while inheriting the core information from one parent class.

10.10a Multiple inheritance



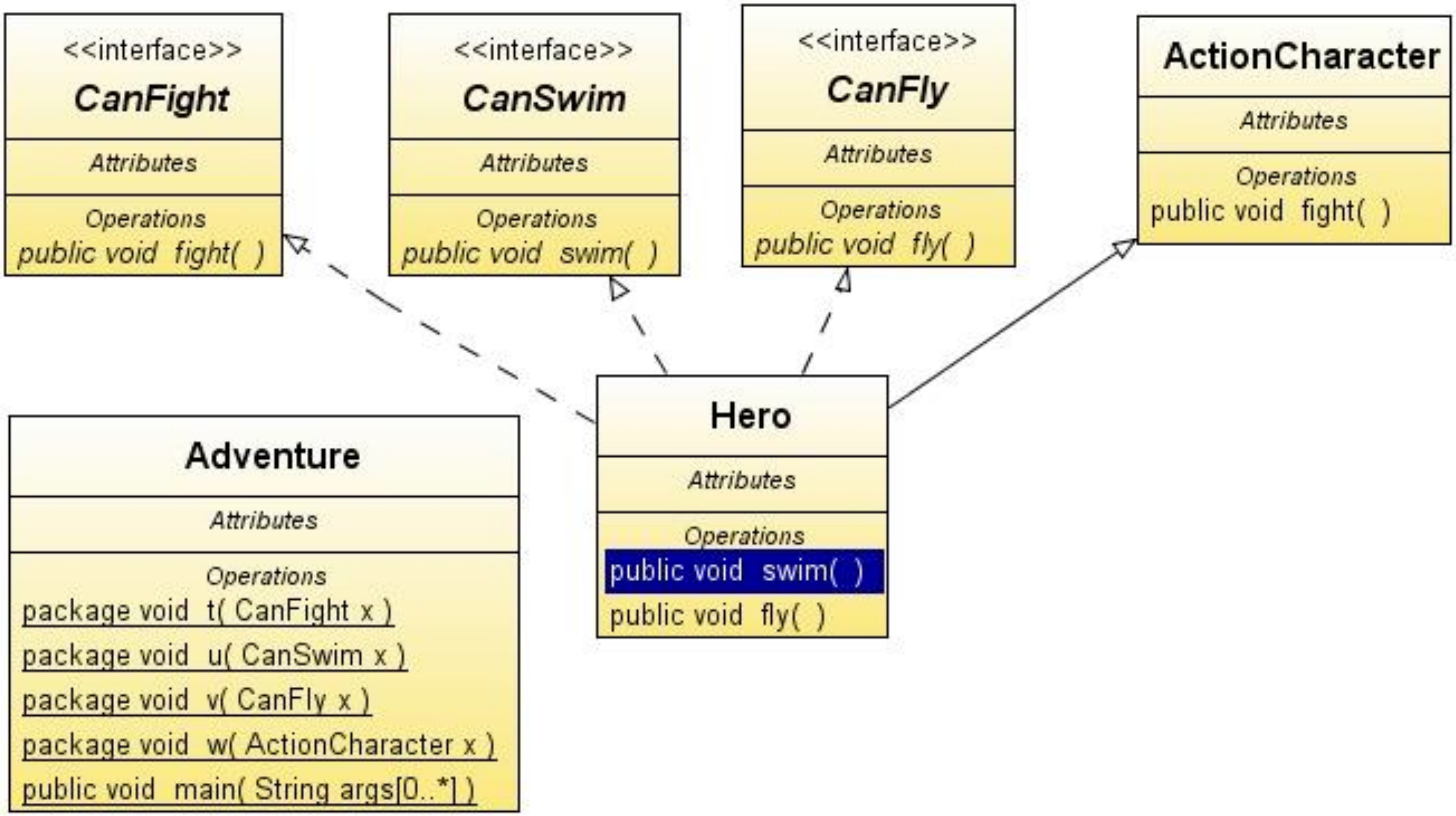
In a derived class, you aren't forced to have a base class that is either an **abstract** or "concrete" (one with no **abstract** methods). If you *do* inherit from a non-**interface**, you can inherit from only one. All the rest of the base elements must be **interface**s. You place all the interface names after the **implements** keyword and separate them with commas.

```
// Multiple interfaces.
import java.util.*;

interface CanFight {
  void fight();
}

interface CanSwim {
  void swim();
}

interface CanFly {
  void fly();
}

class ActionCharacter {
  public void fight() {}
}

class Hero extends ActionCharacter implements CanFight, CanSwim,CanFly
{
  public void swim() {}
  public void fly() {}
}
```

You can see that **Hero** combines the concrete class **ActionCharacter** with the interfaces **CanFight**, **CanSwim**, and **CanFly**.
When you combine a concrete class with interfaces this way, the concrete class must come first, then the interfaces.

```
public class Adventure {

  static void makeTrouble(CanFight x) { x.fight(); }

  static void breakRecord(CanSwim x)  { x.swim(); }

  static void tryFaster(CanFly x)    { x.fly(); }

  static void makeMovie(ActionCharacter x) { x.fight(); }

  public static void main(String[] args)
  {
    Hero hero = new Hero();
    // Hero is an ActionCharacter
    // he also CanFight, CanSwim, CanFly,
    makeTrouble(hero); // Treat hero as a CanFight

    breakRecord(hero); // Treat hero as a CanSwim

    tryFaster(hero);   // Treat hero as a CanFly

    makeMovie(hero);   // Treat hero as an ActionCharacter
  }
}
```

UML diagram - Adventure



10.10a Multiple inheritance

Note that the signature for **fight( )** is the same in the **interface CanFight** and the class **ActionCharacter**, and that **fight( )** is *not* provided with a definition in **Hero**. The rule for an **interface** is that you can inherit from it (as you will see shortly), but then you've got another **interface**. If you want to create an object of the new type, it must be a class with all definitions provided. Even though **Hero** does not explicitly provide a definition for **fight( )**, the definition comes along with **ActionCharacter** so it is automatically provided and it's possible to create objects of **Hero.**

10.10a Multiple inheritance

In class **Adventure**, you can see that there are four methods that take as arguments the various interfaces and the concrete class.

When a **Hero** object is created, it can be passed to any of these methods, which means it is being upcast to each **interface** in turn. Because of the way interfaces are designed in Java, this works without any particular effort on the part of the programmer

10.10b Name collisions when combining interfaces

You can encounter a small **pitfall when implementing multiple interfaces**. The difficulty occurs because overriding, implementation, and overloading get unpleasantly mixed together, and **overloaded methods cannot differ only by return type**. When the last two lines are compiled, the error messages say it all:

InterfaceCollision.java:23: f( ) in C cannot implement f( ) in I1; attempting to use incompatible return type

found : int

required: void

InterfaceCollision.java:24: interfaces I3 and I1 are incompatible; both define f( ), but with different return type

```
interface I1 { void f(); }
interface I2 { int f(int i);}
interface I3 { int f(); }

class C {
    public int f() { return 1; }
}

class C2 implements I1, I2 {
    public void f() {  }
    public int f(int i) {  return 1; } // overloaded
}
class C3 extends C implements I2 {
    public int f(int i) { return 1;  } // overloaded
}

class C4 extends C implements I3 {
// Identical, no problem:
    public int f() { return 1; }
}
// Methods differ only by return type
// Compilation error
class C5 extends C implements I1 { }
// Methods differ only by return type
// Compilation error
interface I4 extends I1, I3 { }
```

10.10c Interface or Abstract class

You might reasonably expect in the following example the output to be "**public f( )**", but a **private** method is automatically **final**, and is also hidden from the derived class.

So **Derived**'s **f( )** in this case is a brand new method—it's not even overloaded since the base-class version of **f( )** isn't visible in **Derived**.

```java
interface F{
    void f();
}
public class PrivateOverride {
    private void f() {
        System.out.println("private f()");
    }
    public static void main(String args[]) {
        PrivateOverride po = new Derived();
        po.f();
        // prints "private f()"
    }
}
class Derived extends PrivateOverride implements F {
    public void f() {
        System.out.println("public f()");
    }
}
```

10.10c Interface or Abstract class

Keep in mind that the core reason for interfaces is shown in the above example: to be able to upcast to more than one base type. However, a second reason for using interfaces is the same as using an **abstract** base class: to prevent the client programmer from making an object of this class and to establish that it is only an interface. This brings up a question: Should you use an **interface** or an **abstract** class? An **interface** gives you the benefits of an **abstract** class *and* the benefits of an **interface**, so if it's possible to create your base class without any method definitions or member variables you should always prefer **interface**s to **abstract** classes. In fact, if you know something is going to be a base class, your first choice should be to make it an **interface**, and only if you're forced to have method definitions or member variables should you change to an **abstract** class, or if necessary a concrete class.

10.10d Interface inheritance

You can easily add new method declarations to an **interface** using inheritance, and you can also combine several **interface**s into a new **interface** with inheritance. In both cases you get a new **interface**, as seen in the following example:

Резюме

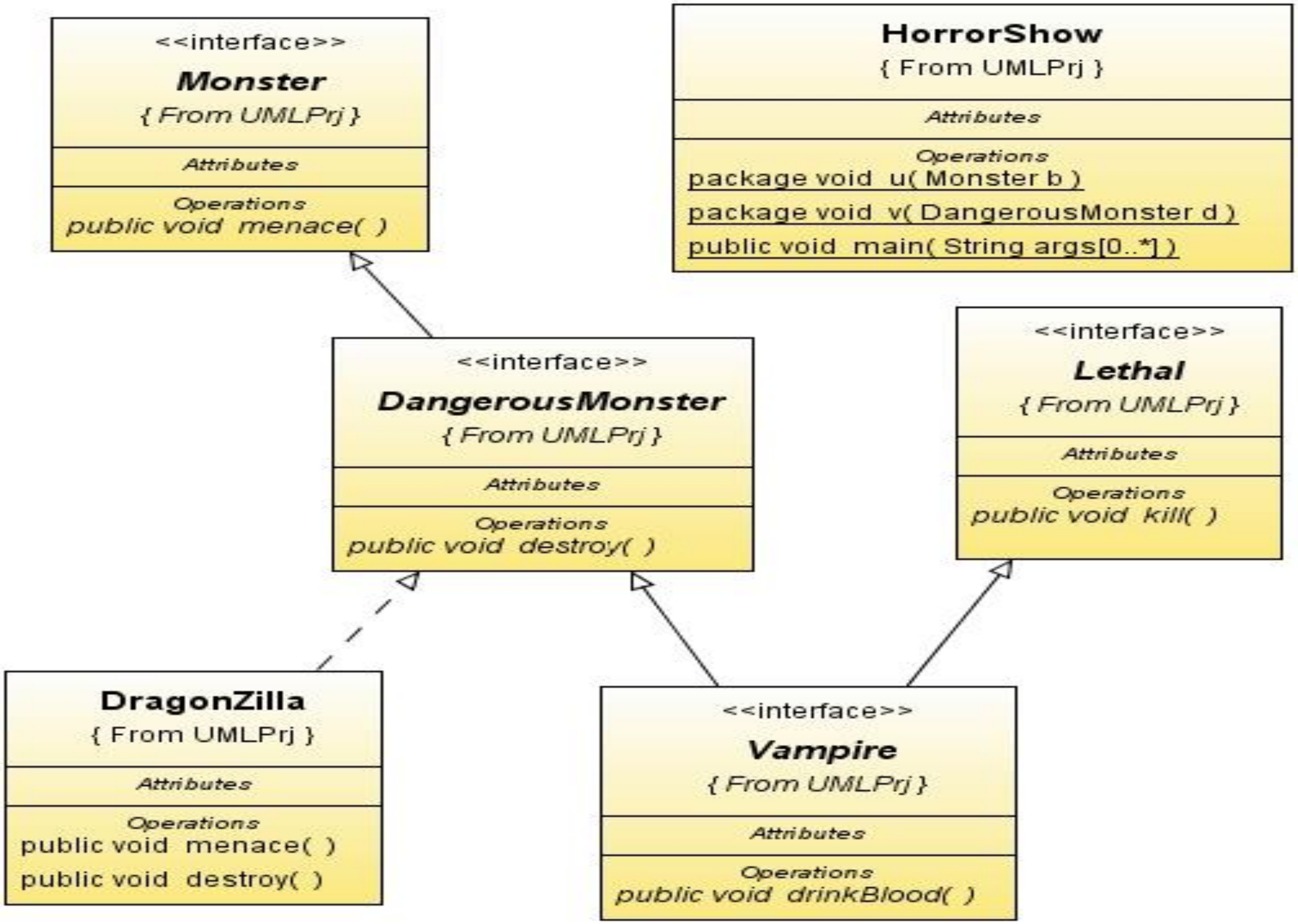
```
// Extending an interface with inheritance.

interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}

class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}

interface Vampire  extends DangerousMonster, Lethal {
  void drinkBlood();
}

class HorrorShow {
  static void feed(Monster b) { b.menace(); }
  static void runAway(DangerousMonster danger) {
    danger.menace();
    danger.destroy();
  }
```

The syntax used in **Vampire** works *only* when inheriting interfaces. Normally, you can use **extends** with only a single class, but since an **interface** can be made from multiple other interfaces, **extends** can refer to multiple base interfaces when building a new **interface**. As you can see, the **interface** names are simply separated with commas.

```
public static void main(String[] args) {
    DragonZilla babyZilla = new DragonZilla();
    feed(babyZilla );
    runAway(babyZilla );
  }
}
```

DangerousMonster is a simple extension to **Monster** that produces a new **interface**. This is implemented in **DragonZilla**

UML diagram - Monster



10.10d Constants with interfaces

Any fields you put into an **interface** are automatically **static** and **final**. Therefore the **interface** may be used for creating groups of constant values

```
// Initializing interface fields with
// non-constant initializers.
import java.util.*;

public interface RandVals {
  int rint = (int)(Math.random() * 10);
  long rlong = (long)(Math.random() * 10);
  float rfloat = (float)(Math.random() * 10);
  double rdouble = Math.random() * 10;
}


// using the interface data members

public class TestRandVals
{
       public static void main(String[] args)
       { System.out.println(RandVals.rint);
         System.out.println(RandVals.rlong);

System.out.println(RandVals.rfloat);
         System.out.println(RandVals.rdouble);
       }
}
```

10.11 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

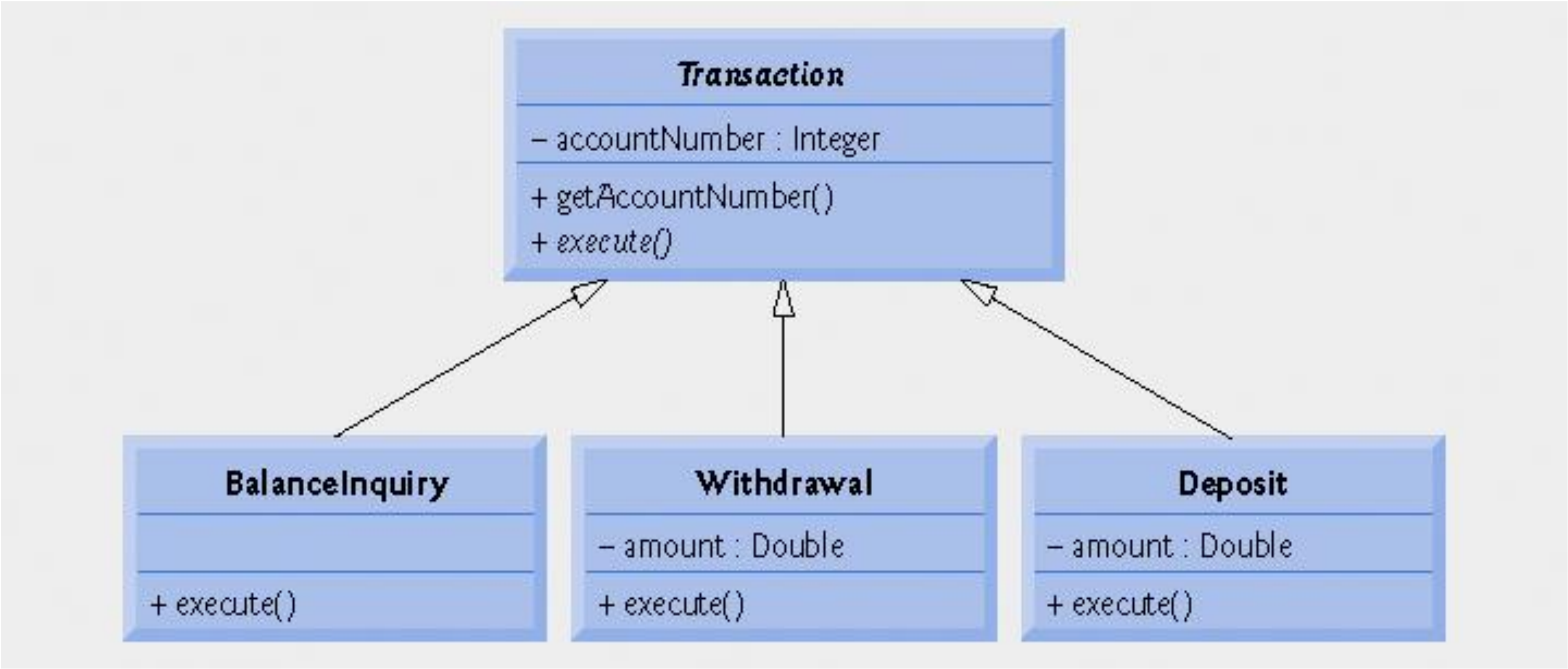


Fig. 10. 20 | **Class diagram modeling generalization of superclass `Transaction` and subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. Note that abstract class names (e.g., `Transaction`) and method names (e.g., `execute` in class `Transaction`) appear in italics.**

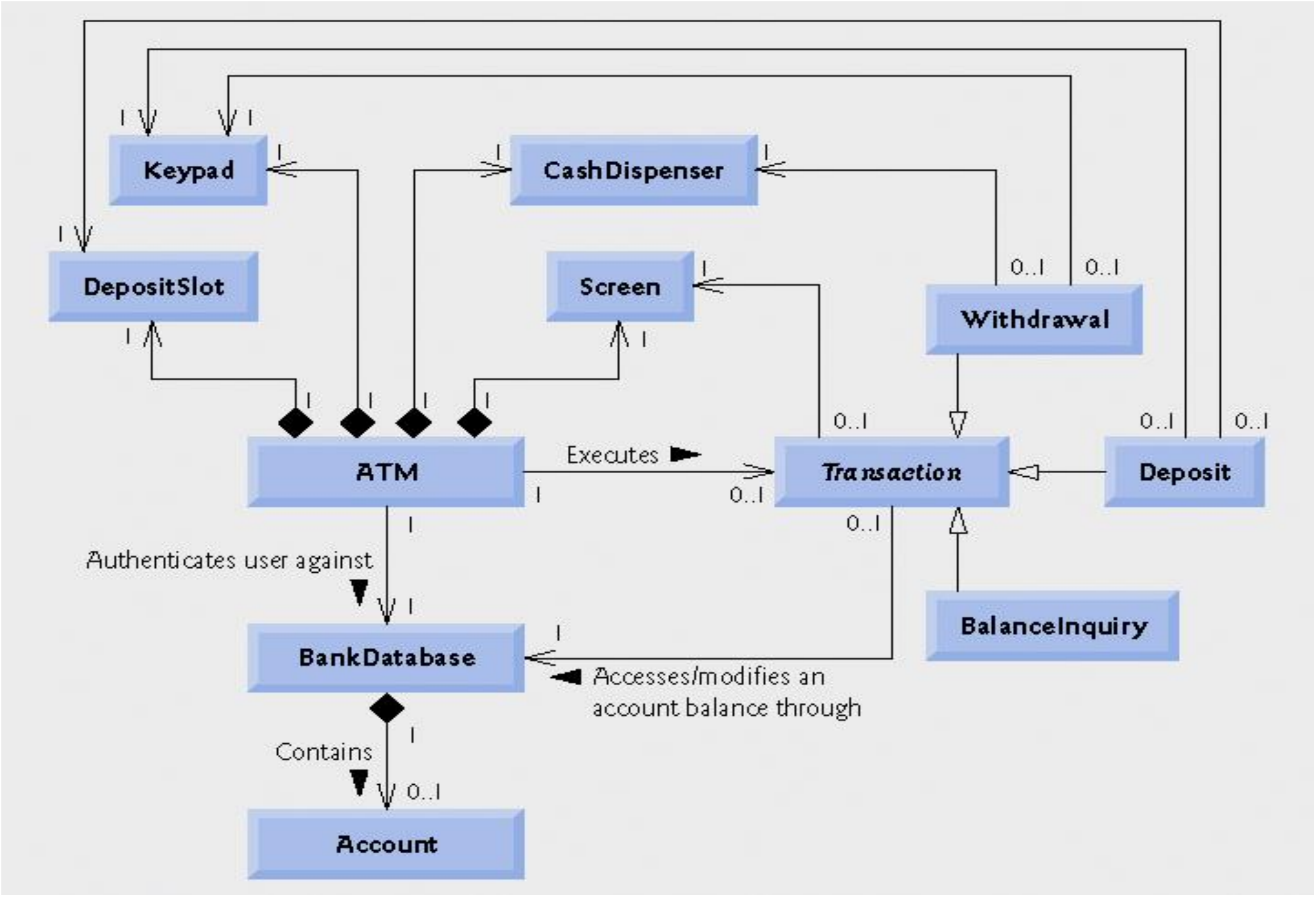


Fig. 10.21 | Class diagram of the ATM system (incorporating inheritance). Note that abstract class names (e.g., `Transaction`) appear in italics.

Software Engineering Observation

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, 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.

10.11 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**

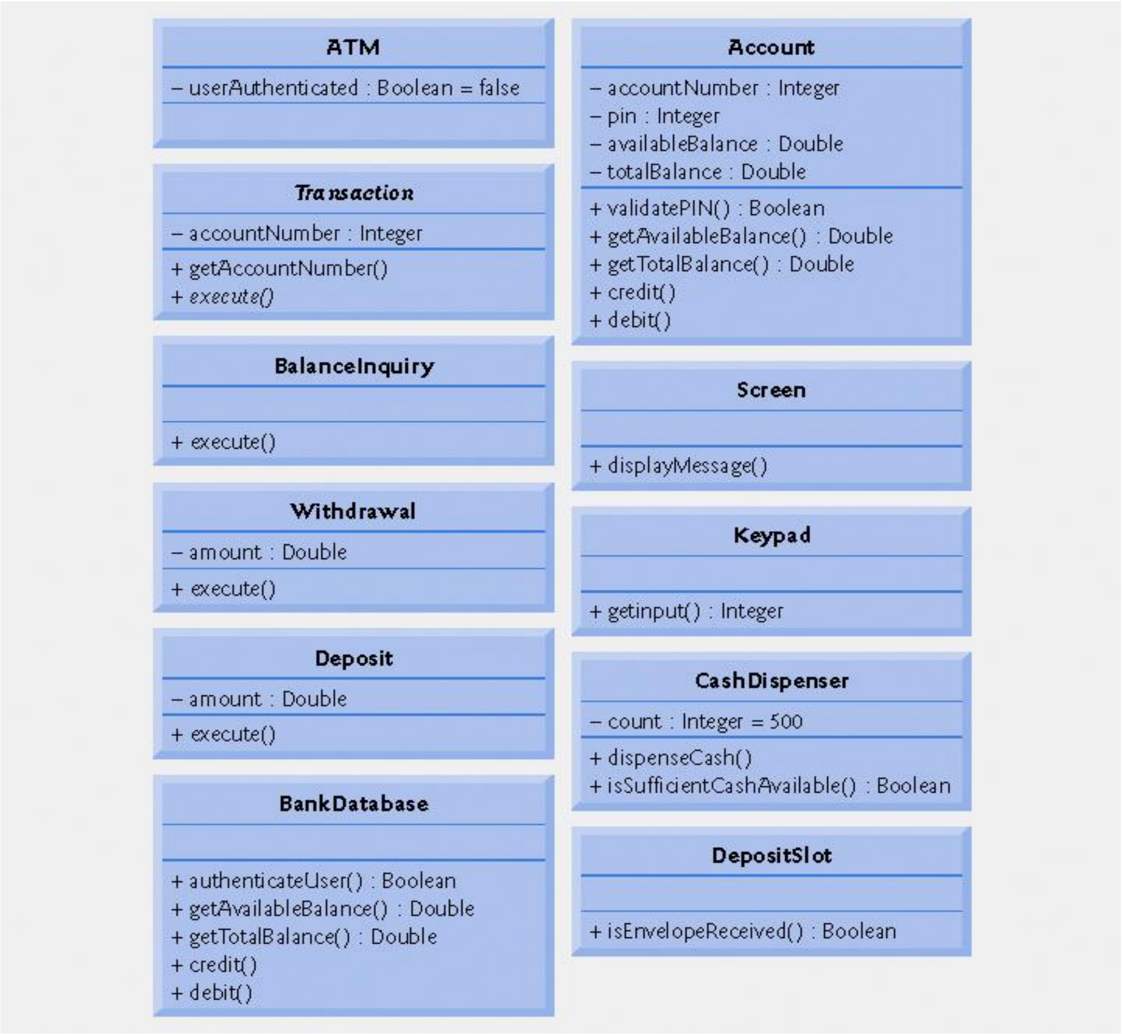


Fig. 10.22 | Class diagram with attributes and operations (incorporating inheritance). Note that abstract class names (e.g., `Transaction`) and method names (e.g., `execute` in class `Transaction`) appear in italic

Outline

```java
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal extends Transaction
{
} // end class Withdrawal
```

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

Withdrawal.java

Outline

Withdrawal.java

```java
// Withdrawal.java
// Generated using the class diagrams in Fig. 10.21 and Fig. 10.22
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
```

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

Software Engineering Observation

Several UML modeling tools convert UML-based designs into Java code and can speed the implementation process considerably.

Outline

```java
// 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

```

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

`Transaction.java`

(1 of 2)

Outline

```java
   // 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**

Transaction.java

10.11.1 Software Engineering Case Study: Enhanced Delegate Pattern

In software engineering, the delegation pattern is a technique where an object outwardly expresses certain behavior but in reality delegates responsibility for implementing that behavior to an associated object in an Inversion of Responsibility. The Delegation pattern is the fundamental abstraction that underpins composition (also referred to as aggregation).

10.11.1 Software Engineering Case Study: Enhanced Delegate Pattern

Assume the class C has method stubs that forward the methods f() and g() to class A. Class C pretends that it has attributes of class A.

```java
class A {
    void f() { System.out.println("A: doing f()"); }
    void g() { System.out.println("A: doing g()"); }
}

class C {
    // delegation
    A a = new A();

    void f() { a.f(); }
    void g() { a.g(); }

    // normal attributes
    X x = new X();
    void y() { /* do stuff */ }
}

public class Main {
    public static void main(String[] args) {
        C c = new C();
        c.f();
        c.g();
    }
}
```

10.11.1 Software Engineering Case Study: Enhanced Delegate Pattern

By using interfaces, delegation can be made more flexible and typesafe.

In this example, class C can delegate to either class A or class B. Class C has methods to switch between classes A and B. Including the implements clauses improves type safety, because each class must implement the methods in the interface.

The main tradeoff is more code

```java
interface I {
    void f();
    void g();
}
class A implements I {
    public void f() { System.out.println("A: doing f()"); }
    public void g() { System.out.println("A: doing g()"); }
}

class B implements I {
    public void f() { System.out.println("B: doing f()"); }
    public void g() { System.out.println("B: doing g()"); }
}

class C implements I {
    // delegation
    I i = new A();
    public void f() { i.f(); }
    public void g() { i.g(); }

    // normal attributes
    void toA() { i = new A(); }
    void toB() { i = new B(); }
}
```

```java
public class Main {
    public static void main(String[] args) {
        C c = new C();
        c.f();
        c.g();
        c.toB();
        c.f();
        c.g();
    }
}
```

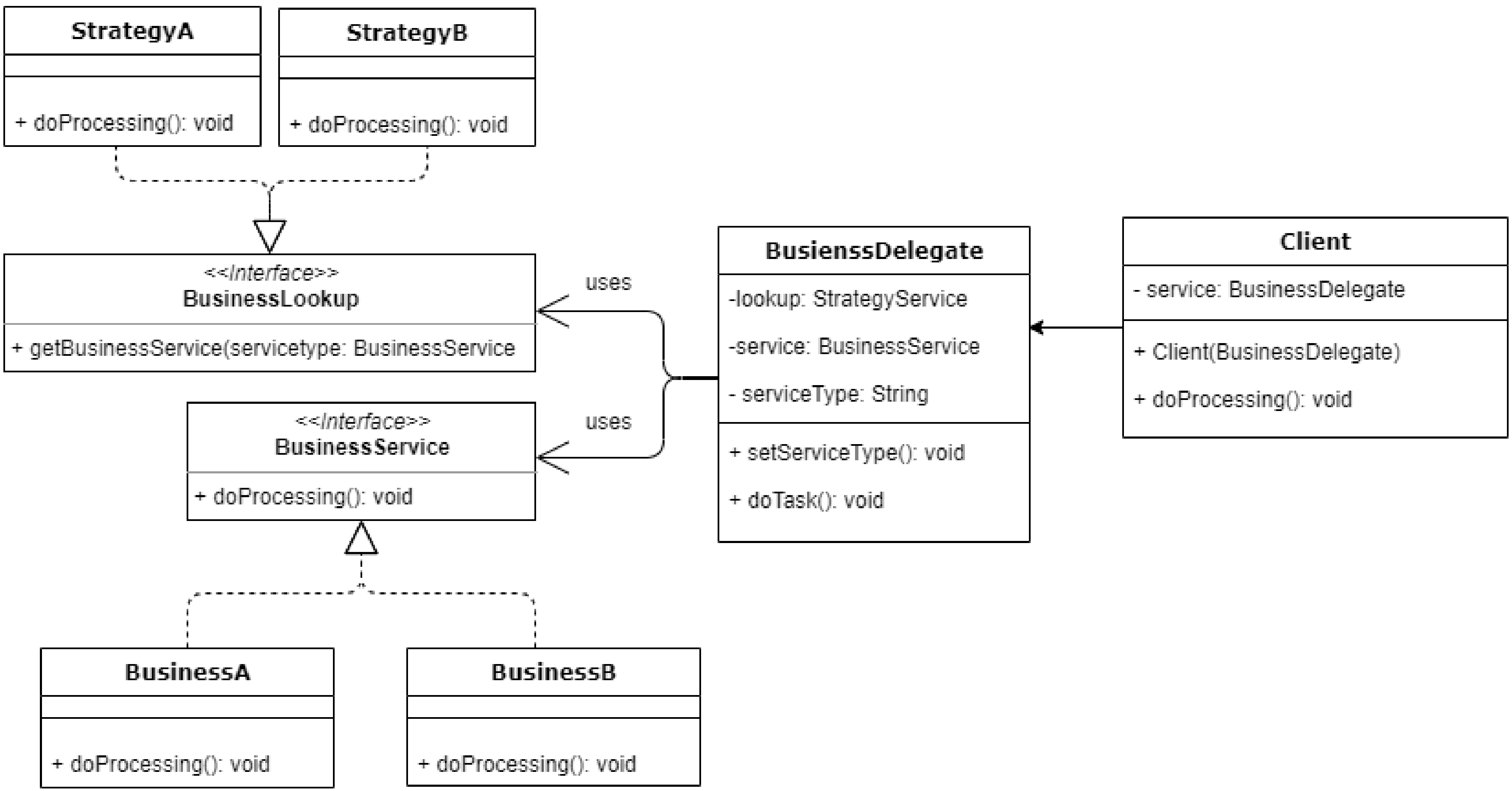
10.12 Business Delegate Pattern

Business Delegate Pattern is used to decouple presentation tier and business tier. It is basically used to **reduce communication or remote lookup functionality** to business tier code in presentation tier code.

In **business tier** we have following entities:

- **Client** - Presentation tier code may be JSP, servlet or UI Java code. Executes task relative to a Business service.
- **Business Delegate** - A single entry point class for client entities to provide access to Business Service methods.
- **LookUp Service** - Lookup service interface. Concrete classes implement the strategy for allocating appropriate business implementation and deliver Business Service object for task processing.
- **Business Service** - Business Service interface. Concrete classes implement this business service to provide actual business implementation logic.

10.12 Business Delegate Pattern



10.12 Business Delegate Pattern

Advantages :

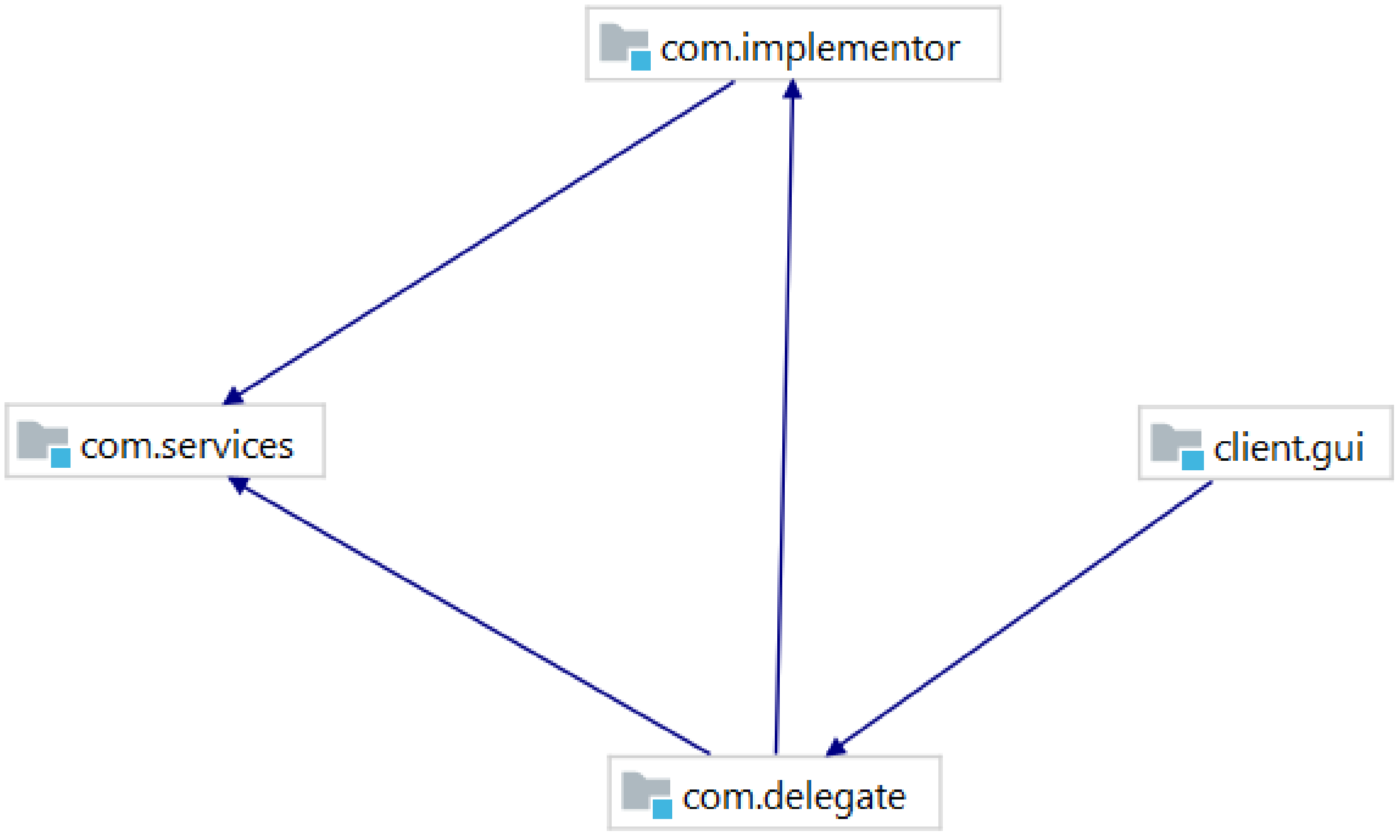
- Business Delegate **reduces coupling** between presentation-tier clients and Business services.
- The Business Delegate **hides the underlying implementation details** of the Business service.

Disadvantages :

- Maintenance due the extra layer that increases the number of classes in the application.

10.12 Business Delegate Pattern

Java modular implementation



10.12 Business Delegate Pattern

Step 1. Create the services in module com.services

```
package com.service;

public interface QuoteService { // maps to the BusinessService interface
   String doProcessing();
}


package com.service;

public interface SelectService {// maps to the BusinessLookup interface
   QuoteService getBusinessService(String serviceType);
}
module com.services {
   exports com.service;  // provide access to the services
}
```

10.12 Business Delegate Pattern

Step 2. Create concrete Service classes in module com.implementor

```
package com.implementor;
// imports omitted for shortness
public class ProfessionalQuoteService implements QuoteService {
   // Concrete classes implementing the BusinessService interface, here QuoteService
  @Override
   public String doProcessing() {
      return "Processing task by invoking Professional Quote Service ";
   }
}
package com.implementor;
// Concrete classes implementing the BusinessLookup interface, here SelectService
public class StrategySelectService implements SelectService {
   @Override
   public QuoteService getBusinessService(String serviceType){
     //  delivers BusinessService objects matching given serviceType
      if(serviceType.equalsIgnoreCase("pro")){   return new ProfessionalQuoteService();      }
      else {        return new SimpleQuoteService();        }
   }
}
```

10.12 Business Delegate Pattern

Step 2. Create concrete Service classes in module com.implementor

Note the module descriptor

```
module com.implementor {

  exports com.implementor; // required by the BusinessDelegate
 // Add Dependency for com.services  module
  requires com.services;        // needed to read the interfaces

// consumes service
  uses com.service.QuoteService;
//provides service implementation
  provides com.service.QuoteService with ProfessionalQuoteService, SimpleQuoteService;

  uses com.service.SelectService;
  provides com.service.SelectService with StrategySelectService;
}
```

10.12 Business Delegate Pattern

Step 3. Create the BusinessDelegate in module com.delegate

```
package com.delegate;
// imports omitted for shortness
public class BusinessDelegate {
   private SelectService lookupService = new StrategySelectService();// default lookup
   private QuoteService businessService;
   private String serviceType;

   public void setServiceType(String serviceType){
      this.serviceType = serviceType;
   }

   public String doTask(){
      businessService = lookupService.getBusinessService(serviceType);
      String output = businessService.doProcessing(); // execute Task delegated by Client
      return output;
   }
}
```

10.12 Business Delegate Pattern

Step 3. Create the BusinessDelegate in module com.delegate

```
module com.delegate {
 // Add Dependency for all the required modules
  requires com.services;  // reads interfaces
  requires com.implementor; // reads BusinessLookup default implementation

  uses com.service.QuoteService;  // uses service
  uses com.service.SelectService; // uses service

  exports  com.delegate; // required by Client
}
```

10.12 Business Delegate Pattern

Step 4. Create the Client in module client.gui

```
package client.gui;
// imports omitted for shortness

public class Client {

   private BusinessDelegate businessService;

   public Client(BusinessDelegate businessService){
      this.businessService  = businessService;
   }

   public String doTask(){ // method execution delegated to BusinessService object!!
      return businessService.doTask();
   }
}
```

10.12 Business Delegate Pattern

Step 4. Create the Client in module client.gui

```
package client.gui; // Client Presentation tier
// imports omitted for shortness
public class ClientUI extends Application {
   private BusinessDelegate businessDelegate = new BusinessDelegate();
   private Client client = new Client(businessDelegate);
   public static void main(String[] args) {
      Application.launch(args);
   }
   @Override
   public void start(Stage primaryStage)     throws Exception {
     // JavaFX setup omitted for shortness
      businessDelegate.setServiceType("pro");
      Label labelPro = new Label(client.doTask()); // delegates Task execution
      businessDelegate.setServiceType("simple");
      Label labelSimple = new Label(client.doTask()); // delegates Task execution
       // JavaFX setup omitted for shortness
   }
}
```

10.12 Business Delegate Pattern

Step 4. Create the Client in module client.gui

```
// client.gui module descriptor

module client.gui {
   requires javafx.fxml;
   requires  javafx.controls;

   requires  com.delegate; // add Dependency for this module
  // no FXML used here
   exports client.gui to javafx.graphics;
}
```

10.13 Bad Use of Override Methods

One can **override** the operations of a superclass with completely new meanings

Example:

```
public class SuperClass {
  public int add (int a, int b) { return a+b; }
  public int subtract (int a, int b) { return a-b; }
}
public class SubClass extends SuperClass {
  public int add (int a, int b) { return a-b; }
  public int subtract (int a, int b) { return a+b; }
}
```

We have redefined addition as subtraction and subtraction as addition!!

10.13 Bad Use of Override Methods

We have **delivered a car with** software that allows to operate an **on-board stereo system**

- A customer wants to have **software for a cheap stereo system to be sold by a discount store chain**

Dialog between project manager and developer:

- Project Manager:
  - „Reuse the existing car software. Don't change this software, make sure there are no hidden surprises. There is no additional budget, deliver tomorrow!"
- Developer:
  - „OK, we can easily create a subclass BoomBox inheriting the operations from the existing Car software"
  - „And we **override all method implementations** from Car that **have nothing to do with playing music with empty bodies**!"

What we have and what we want

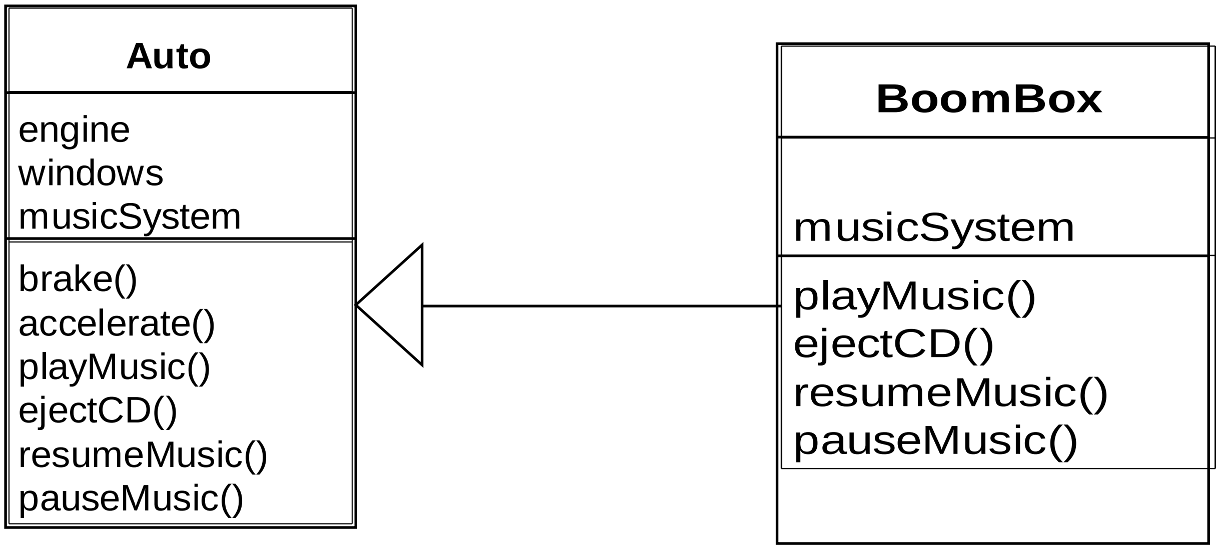
Auto
engine windows musicSystem
brake() accelerate() playMusic() ejectCD() resumeMusic() pauseMusic()

ExistingProduct!

BoomBox
musicSystem
playMusic() ejectCD() resumeMusic() pauseMusic()

New Product!

What we do to save money and time



Existing Product:

```
public class Auto {
  public void drive() {…}
  public void brake() {…}
  public void accelerate() {…}
  public void playMusic() {…}
  public void ejectCD() {…}
  public void resumeMusic() {…}
  public void pauseMusic() {…}
}
```

New Product:

```
public class Boombox extends
Auto {
   public void drive() {};
   public void brake() {};
   public void accelerate() {};
}
```

10.13a Contraction

Contraction: Implementations of methods in the super class are overwritten with empty bodies in the subclass to make the super class operations "invisible"

✓Contraction is a special type of inheritance

✓It should be avoided at all costs (violates the **Liskov** substitution principle) , but is used often.

10.13b The Liskov substitution principle

Substitutability is **a principle in object-oriented programming**. It states that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may be ***substituted*** for objects of type T) **without altering any of the desirable properties of that program** (correctness, task performed, etc.).

It was initially introduced by **Barbara Liskov** in a 1987 conference keynote address entitled *Data abstraction and hierarchy*. It is **a semantic rather than merely syntactic relation** because it intends to guarantee semantic interoperability of object types in a hierarchy of inheritance.

10.13c Contraction should be avoided

A contracted s**ubclass delivers the desired functionality** expected by the client, but:

- The interface contains operations that make no sense for this class
- What is the meaning of the operation brake() for a BoomBox?

The **subclass does not fit into the taxonomy**

A BoomBox is not a special form of Auto

The subclass violates **Liskov's Substitution** Principle:

- I cannot replace Auto with BoomBox to drive to work.

Задачи

Problem 1.

Create a base class with an **abstract print( )** method that is overridden in a derived class. The overridden version of the method prints the value of an **int** variable defined in the derived class. At the point of definition of this variable, give it a nonzero value. In the base-class constructor, call this method. In **main( )**, create an object of the derived type, and then call its **print( )** method. Explain the results.

Problem 2.

Create an **abstract** class with no methods. Derive a class and add a method. Create a **static** method that takes a reference to the base class, downcasts it to the derived class, and calls the method.

In **main( )**, demonstrate that it works. Now put the **abstract** declaration for the method in the base class, thus eliminating the need for the downcast.

Задачи

Problem 3.

Create a base class with two methods. In the first method, call the second method. Inherit a class and override the second method.

Create an object of the derived class, upcast it to the base type, and call the first method. Explain what happens.