Chapter 10
Object-Oriented Programming
Part 3:
Inheritance, Polymorphism, and Interfaces

Topics
• Inheritance Concepts
• Inheritance Design
  – Inherited Members of a Class
  – Subclass Constructors
  – Adding Specialization to the Subclass
  – Overriding Inherited Methods
• The protected Access Modifier
• Abstract Classes and Methods
• Polymorphism
• Interfaces

Inheritance Concepts
• A common form of reuse of classes is inheritance.
• We can organize classes into hierarchies of functionality.
• The class at the top of the hierarchy (superclass) defines instance variables and methods common to all classes in the hierarchy.
• We derive a subclass, which inherits behavior and fields from the superclass.

A Sample Vehicle Hierarchy
• This hierarchy is depicted using a Unified Modeling Language (UML) diagram.
• In UML diagrams, arrows point from the subclass to the superclass.
Superclasses and Subclasses

• A superclass can have multiple subclasses.

• Subclasses can be superclasses of other subclasses.

• A subclass can inherit directly from only one superclass.

• All classes inherit from the Object class.

A big advantage of inheritance is that we can write common code once and reuse it in subclasses.

– Generalization

• A subclass can define new methods and instance variables, some of which may override (hide) those of a superclass.

– Specialization

Specifying Inheritance

• The syntax for defining a subclass is to use the extends keyword in the class header, as in

```java
accessModifier class SubclassName extends SuperclassName
{
    // class definition
}
```

• The superclass name specified after the extends keyword is called the direct superclass.

• As mentioned, a subclass can have many superclasses, but only one direct superclass.

An Applet Hierarchy

• When we wrote an applet, we defined a subclass.

• We say that inheritance implements an "is a" relationship, in that a subclass object "is a" superclass object as well.

• Thus, RollABall "is a" JApplet (its direct superclass), Applet, Panel, Container, Component, and Object.

• RollABall begins with more than 275 methods and 15 fields inherited from its 6 superclasses.
The Bank Account Hierarchy

• The BankAccount class is the superclass.

  – Instance variables:
    • balance (double)
    • MONEY (final DecimalFormat)

  – Methods:
    • Default and overloaded constructors
    • deposit and withdraw methods
    • balance accessor
    • toString

• See Example 10.1 BankAccount.java (next slide)

BankAccount.java 1/3

import java.text.DecimalFormat;
public class BankAccount {
    public final DecimalFormat MONEY = new DecimalFormat( "$#,##0.00" );
    private double balance;
    public BankAccount( ) {
        balance = 0.0;
    }
    public BankAccount( double startBalance ) {
        deposit( startBalance );
    }
    public double getBalance( ) {
        return balance;
    }
    public void deposit( double amount ) {
        if ( amount >= 0.0 )
            balance += amount;
        else
            System.err.println( "Deposit amount must be positive." );
    }
    public void withdraw( double amount ) {
        if ( amount >= 0.0 && amount <= balance )
            balance -= amount;
        else
            System.err.println( "Withdrawal amount must be positive and cannot be greater than balance" );
    }
    public String toString( ) {
        return ( "balance is " + MONEY.format( balance ) );
    }
}

BankAccount.java 2/3

BankAccount.java 3/3
The CheckingAccount Class

• We derive the CheckingAccount subclass from BankAccount:

```java
public class CheckingAccount extends BankAccount {
}
```

• A subclass inherits all the public members of a superclass. Thus, the CheckingAccount class inherits
  – the MONEY instance variable
  – The getBalance, deposit, withdraw, and toString methods
• See Example 10.3 CheckingAccountClient.java (next slide)

private Members

• Superclass members declared as private are NOT inherited, although they are part of the subclass.

• Thus, the balance instance variable is allocated to all CheckingAccount objects, but methods of the
  CheckingAccount class cannot directly access balance.

• To set or get the value of balance, the CheckingAccount methods must call the withdraw, deposit, or getBalance
  methods of the Superclass BankAccount.

• This simplifies maintenance because the BankAccount class enforces the data validation rules for balance.

protected Members

• protected members are inherited by subclasses (like public members), while still being hidden
  from client classes (like private members).

• Also, any class in the same package as the superclass can directly access a protected field, even if that class is not a subclass.

• Disadvantage:
  – Because more than one class can directly access a protected field, protected access compromises
    encapsulation and complicates maintenance of a program.
  – For that reason, we prefer to use private, rather than protected, for our instance variables.

CheckingAccountClient.java

```java
public class CheckingAccountClient {
    public static void main(String[] args) {
        CheckingAccount c1 = new CheckingAccount();
        System.out.println( "New checking account: " + c1 );
        c1.deposit(350.75);
        System.out.println("After depositing $350.75: " + c1);
        c1.withdraw(200.25);
        System.out.println("After withdrawing $200.25: " + c1);
    }
}
```
### Inheritance Rules

<table>
<thead>
<tr>
<th>Superclass Members</th>
<th>Inherited by subclass?</th>
<th>Directly Accessible by Subclass?</th>
<th>Directly Accessible by Client of Subclass?</th>
</tr>
</thead>
<tbody>
<tr>
<td>public fields</td>
<td>yes</td>
<td>yes, by using field name</td>
<td>yes</td>
</tr>
<tr>
<td>public methods</td>
<td>yes</td>
<td>yes, by calling method from subclass methods</td>
<td>yes</td>
</tr>
<tr>
<td>protected fields</td>
<td>yes</td>
<td>yes, by using field name</td>
<td>no, must call accessors and mutators</td>
</tr>
<tr>
<td>protected methods</td>
<td>yes</td>
<td>yes, by calling method from subclass methods</td>
<td>no</td>
</tr>
<tr>
<td>private fields</td>
<td>no</td>
<td>no, must call accessors and mutators</td>
<td>no</td>
</tr>
<tr>
<td>private methods</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

### Subclass Constructors

- Constructors are not inherited.
- However, the subclass can call the constructors of the superclass to initialize inherited fields.
- **Implicit** invocation
  - The default constructor of the subclass automatically calls the default constructor of the superclass
- For **explicit** invocation, use this syntax:

  ```java
  super( argument list );
  ```
  
  If used, this statement must be the first statement in the subclass constructor

### CheckingAccount Constructors

```java
public CheckingAccount() {
    // optional explicit call
    // to BankAccount default constructor
    super();
}

public CheckingAccount(double startBalance) {
    // explicit call to BankAccount
    // overloaded constructor
    super( startBalance );
}
```

- **See Examples 10.4 (BankingAccount.java V2 & 10.5 CheckingAccount.java V2)**

### BankAccount.java Version 2

```java
import java.text.DecimalFormat;

class BankAccount {
    public final DecimalFormat MONEY = new DecimalFormat( "$#,##0.00" );
    private double balance;

    public BankAccount() {
        balance = 0.0;
        System.out.println( "In BankAccount default constructor" );
    }

    public BankAccount(double startBalance) {
        if ( balance >= 0.0 ) balance = startBalance;
        else balance = 0.0;
        System.out.println( "In BankAccount overloaded constructor" );
    }

    public String toString() {
        return ( "balance is " + MONEY.format( balance ) );
    }
}
CheckingAccount.java Version 2

```java
public class CheckingAccount extends BankAccount {
    public CheckingAccount() {
        super(); // optional, call BankAccount constructor
        System.out.println("In CheckingAccount " + "default constructor");
    }
    public CheckingAccount(double startBalance) {
        super(startBalance); // call BankAccount constructor
        System.out.println("In CheckingAccount " + "overloaded constructor");
    }
}
```

Common Error Trap

- An attempt by a subclass to directly access a `private` field or call a `private` method defined in a superclass will generate a compiler error.
- To set initial values for `private` variables, call the appropriate constructor of the direct superclass.
- For example, this statement in the overloaded `CheckingAccount` class constructor calls the overloaded constructor of the `BankAccount` class:

  ```java
  super(startBalance);
  ```

Software Engineering Tip

Overloaded constructors in a subclass should explicitly call the direct superclass constructor to initialize the fields in its superclasses.

Inheritance Rules for Constructors

<table>
<thead>
<tr>
<th>Superclass Members</th>
<th>Inherited by subclass?</th>
<th>Directly Accessible by Subclass?</th>
<th>Directly Accessible by Client of Subclass Using a Subclass Reference?</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructors</td>
<td>no</td>
<td>yes, using <code>super(arg list)</code> in a subclass constructor</td>
<td>no</td>
</tr>
</tbody>
</table>
Adding Specialization

- A subclass can define new fields and methods.
- Our `CheckingAccount` class adds
  - these instance variables:
    - `monthlyFee`, a `double`
    - `DEFAULT_FEE`, a `double` constant
  - these methods:
    - `setMonthlyFee`, the accessor
    - `getMonthlyFee`, the mutator
    - `applyMonthlyFee`, which charges the monthly fee to the account.

The `applyMonthlyFee` Method

- Because `balance` is `private` in the `BankAccount` class, the `applyMonthlyFee` method calls the `withdraw` method to subtract the monthly fee from the balance:

  ```java
  public void applyMonthlyFee( )
  {
      withdraw( monthlyFee );
  }
  ```

- See Examples 10.7 (CheckingAccount V3 & 10.8 CheckingAccountClient V3)

```java
public class CheckingAccount extends BankAccount {
    public final double DEFAULT_FEE = 5.00;
    private double monthlyFee;
    public CheckingAccount( ) {
        super( ); // optional
        monthlyFee = DEFAULT_FEE;
    }
    public CheckingAccount( double startBalance, double startMonthlyFee ) {
        super( startBalance ); // call BankAccount constructor
        setMonthlyFee( startMonthlyFee );
    }
    public void applyMonthlyFee( ) {
        withdraw( monthlyFee );
    }
    public double getMonthlyFee( ) {
        return monthlyFee;
    }
    public void setMonthlyFee( double newMonthlyFee ) {
        if ( monthlyFee >= 0.0 )
            monthlyFee = newMonthlyFee;
        else
            System.err.println( "Monthly fee cannot be negative" );
    }
}
```
CheckingAccountClient.java V3

```java
public class CheckingAccountClient {
    public static void main(String[] args) {
        CheckingAccount c3 = new CheckingAccount(100.00, 7.50);
        System.out.println("New checking account:
" + c3.toString() + "; monthly fee is 
" + c3.getMonthlyFee());
        c3.applyMonthlyFee(); // charge the fee to the account
        System.out.println("After charging monthly fee:
" + c3.toString() + "; monthly fee is 
" + c3.getMonthlyFee());
    }
}
```

Software Engineering Tip

The superclasses in a class hierarchy should contain fields and methods common to all subclasses.

The subclasses should add specialized fields and methods.

Overriding Inherited Methods

- A subclass can override (or replace) an inherited method by providing a new version of the method.
- The API of the new version must match the inherited method.
- When the client calls the method, it will call the overridden version.
- The overridden method is invisible to the client of the subclass, but the subclass methods can still call the overridden method using this syntax:

```
super.methodName( argument list )
```

The `toString` Method

- The `toString` method in the `CheckingAccount` class overrides the `toString` method in the `BankAccount` class.
- The subclass version call the superclass version to return `balance`.

```
public String toString() {
    return super.toString()
        + "; monthly fee is " + MONEY.format(monthlyFee);
}
```

- See Examples 10.9 & 10.10
Inheritance Rules for Overridden Methods

<table>
<thead>
<tr>
<th>Superclass Members</th>
<th>Inherited by subclass?</th>
<th>Directly Accessible by Subclass?</th>
<th>Directly Accessible by Client of Subclass Using a Subclass Reference?</th>
</tr>
</thead>
<tbody>
<tr>
<td>public or protected inherited methods that have been overridden in the subclass</td>
<td>no</td>
<td>yes, using super.methodName(arg list)</td>
<td>no</td>
</tr>
</tbody>
</table>

Common Error Trap

• Do not confuse overriding a method with overloading a method.
  – Overriding a method:
    • A subclass provides a new version of that method (same signature), which hides the superclass version from the client.
  – Overloading a method:
    • A class provides a version of the method, which varies in the number and/or type of parameters (different signature). A client of the class can call any of the public versions of overloaded methods.

The protected Access Modifier

• Declaring fields as private preserves encapsulation.
  – Subclass methods call superclass methods to set the values of the fields, and the superclass methods enforce the validation rules for the data.
  – But calling methods incurs processing overhead.
• Declaring fields as protected allows them to be accessed directly by subclass methods.
  – Classes outside the hierarchy and package must use accessor and mutator for protected fields.

protected fields: Tradeoffs

• Advantage:
  – protected fields can be accessed directly by subclasses, so there is no method-invocation overhead.
• Disadvantage:
  – Maintenance is complicated because the subclass also needs to enforce validation rules.
• Recommendation:
  – Define protected fields only when high performance is necessary.
  – Avoid directly setting the values of protected fields in the subclass.
• See Examples 10.11, 10.12, & 10.13
abstract Classes and Methods

• An abstract class is a class that is not completely implemented.

• Usually, the abstract class contains at least one abstract method.
  – An abstract method specifies an API but does not provide an implementation.
  – The abstract method is used as a pattern for a method the subclasses should implement.

More on abstract Classes

• An object reference to an abstract class can be declared.
  – We use this capability in polymorphism, discussed later.
• An abstract class cannot be used to instantiate objects (because the class is not complete).
• An abstract class can be extended.
  – subclasses can complete the implementation and objects of those subclasses can be instantiated.

Defining an abstract class

• To declare a class as abstract, include the abstract keyword in the class header:

  accessModifier abstract class ClassName
  {
    // class body
  }

Defining an abstract Method

• To declare a method as abstract, include the abstract keyword in the method header:

  accessModifier abstract returnType
  methodName( argument list );

• Note:
  – The semicolon at the end of the header indicates that the method has no code.
  – We do not use open and closing curly braces
Example Hierarchy

• We can define a Figure hierarchy.
• The superclass is Figure, which is abstract. (In the UML diagram, Figure is set in italics to indicate that it is abstract.)
• We will derive two subclasses: Circle and Square.

The Figure Class

```java
public abstract class Figure {
    private int x;
    private int y;
    private Color color;
    // usual constructors, accessors, // and mutators

    // abstract draw method
    public abstract void draw(Graphics g);
}
```

• All classes in the hierarchy will have an (x, y) coordinate and color. Subclasses will implement the draw method.

Subclasses of abstract Classes

• A subclass of an abstract class can implement all, some, or none of the abstract methods.
• If the subclass does not implement all of the abstract methods, it must also be declared as abstract.
• Our Circle subclass adds a radius instance variable and implements the draw method.
• Our Square subclass adds a length instance variable and implements the draw method.
• See Examples 10.15, 10.16, 10.17, & 10.18

Restrictions for Defining abstract Classes

• Classes must be declared abstract if the class contains any abstract methods.

• abstract classes can be extended.

• An object reference to an abstract class can be declared.

• abstract classes cannot be used to instantiate objects.
Restrictions for Defining \textit{abstract} Methods

- \textit{abstract} methods can be declared only within an \textit{abstract} class.
- An \textit{abstract} method must consist of a method header followed by a semicolon.
- \textit{abstract} methods cannot be called.
- \textit{abstract} methods cannot be declared as \textit{private} or \textit{static}.
- A constructor cannot be declared \textit{abstract}.

Polymorphism

- An important concept in inheritance is that an object of a subclass is also an object of any of its superclasses.
- That concept is the basis for an important OOP feature, called \textbf{polymorphism}.
- Polymorphism simplifies the processing of various objects in the same class hierarchy because we can use the same method call for any object in the hierarchy using a superclass object reference.

Polymorphism Requirements

- To use polymorphism, these conditions must be true:
  1. the classes are in the same hierarchy.
  2. all subclasses override the same method.
  3. a subclass object reference is assigned to a superclass object reference.
  4. the superclass object reference is used to call the method.

Example

- Example 10.19 shows how we can simplify the drawing of \textit{Circle} and \textit{Square} objects.
- We instantiate a \textit{Figure ArrayList} and add \textit{Circle} and \textit{Square} objects to it.

```java
ArrayList<Figure> figuresList = new ArrayList<Figure>();
figuresList.add( new Square( 150, 100, Color.BLACK, 40 ) );
figuresList.add( new Circle( 160, 110, Color.RED, 10 ) );
```

- In the \textit{paint} method, we call \textit{draw} this way:
  ```java
  for ( Figure f : figuresList )
    f.draw( g );
  ```
Polymorphism Conditions

- Example 10.19 shows that we have fulfilled the conditions for polymorphism:
  1. The Figure, Circle, and Square classes are in the same hierarchy.
  2. The non-abstract Circle and Square classes implement the draw method.
  3. We assigned the Circle and Square objects to Figure references.
  4. We called the draw method using Figure references.

Interfaces

- A class can inherit directly from only one class, that is, a class can extend only one class.
- To allow a class to inherit behavior from multiple sources, Java provides the interface.
- An interface typically specifies behavior that a class will implement. Interface members can be any of the following:
  - classes
  - constants
  - abstract methods
  - other interfaces

Interface Syntax

- To define an interface, use the following syntax:
  ```java
  accessModifier interface InterfaceName
  {
  // body of interface
  }
  ```
- All interfaces are abstract; thus, they cannot be instantiated. The abstract keyword, however, can be omitted in the interface definition.

Finer Points of Interfaces

- An interface's fields are public, static, and final.
  - These keywords can be specified or omitted.
- When you define a field in an interface, you must assign a value to the field.
- All methods within an interface must be abstract, so the method definition must consist of only a method header and a semicolon.
  - The abstract keyword also can be omitted from the method definition.
Inheriting from an Interface

- To inherit from an interface, a class declares that it implements the interface in the class definition, using the following syntax:
  ```java
  accessModifier class ClassName
  extends SuperclassName
  implements Interface1, Interface2, ...
  ```
- The `extends` clause is optional.
- A class can implement 0, 1, or more interfaces.
- When a class implements an interface, the class must provide an implementation for each method in the interface.

Example

1. Define an abstract class `Animal` with one abstract method (See Example 10.22):
   ```java
   public abstract void draw( Graphics g );
   ```
2. Define a `Moveable` interface with one abstract method:
   ```java
   public interface Moveable
   {
   int FAST = 5; // static constant
   int SLOW = 1; // static constant
   void move( ); // abstract method
   }
   ```

Derived Classes

- `TortoiseRacer` class
  - `extends Animal` class
  - implements `Moveable` interface
  - implements `draw` and `move` methods
- `TortoiseNonRacer` class
  - `extends Animal` class
  - (does not implement `Moveable` interface)
  - implements `draw` method only
- See Examples 10.21, 10.22, 10.23, & 10.24

Backup
Final Methods and Classes

- A method that is declared final can’t be overridden
- A class that is declared final can’t be a superclass
  - All methods in a final class are final

Abstract Classes Example

- Shape
  - Defines all methods that are common to shapes
- Point
  - Inherits these methods
- Circle
  - Inherits some and overrides some methods
- Cylinder
  - Inherits some and overrides some methods

Shape Example: Shape Class

```java
public abstract class Shape extends Object {
    // return shape's area, overridden when it make sense
    public double area() {
        return 0.0;
    }
    // return shape's volume, overridden when it make sense
    public double volume() {
        return 0.0;
    }
    // abstract method must be overridden by all concrete
    // subclasses to return appropriate shape name
    public abstract String getName();
} // end class Shape
```
Shape Example: Point Class 1/2

```java
public class Point extends Shape {
    protected int x, y; // coordinates of the Point

    public Point() {
        setPoint(0, 0);
    }

    public Point(int xCoordinate, int yCoordinate) {
        setPoint(xCoordinate, yCoordinate);
    }

    public void setPoint(int xCoordinate, int yCoordinate) {
        x = xCoordinate;
        y = yCoordinate;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

Shape Example: Point Class 2/2

```java
public int getY() {
    return y;
}
```

Shape Example: Circle Class 1/2

```java
public class Circle extends Point { // inherits from Point
    protected double radius;

    public Circle() {
        // implicit call to superclass constructor here
        setRadius(0);
    }

    public Circle(double circleRadius, int xCoordinate, int yCoordinate) {
        // call superclass constructor
        super(xCoordinate, yCoordinate);
        setRadius(circleRadius);
    }

    public void setRadius(double circleRadius) {
        radius = (circleRadius >= 0 ? circleRadius : 0);
    }

    public double getRadius() {
        return radius;
    }

    public String toString() {
        return "Center = " + super.toString() + "; Radius = " + radius;
    }

    public String getName() {
        return "Circle";
    }
}
```

Shape Example: Circle Class 2/2

```java
public double area() {
    return Math.PI * radius * radius;
}
```
public class Cylinder extends Circle {
    protected double height;  // height of Cylinder
    public Cylinder() {
        setHeight(0);
    }
    public Cylinder(double cylinderHeight, double cylinderRadius, int xCoordinate, int yCoordinate) {
        super(cylinderRadius, xCoordinate, yCoordinate);
        setHeight(cylinderHeight);
    }
    public void setHeight(double cylinderHeight) {
        height = (cylinderHeight >= 0 ? cylinderHeight : 0);
    }
    public double getHeight() {
        return height;
    }
    // create shapes
    Point point = new Point(7, 11);
    Circle circle = new Circle(3.5, 22, 8);
    Cylinder cylinder = new Cylinder(10, 3.3, 10, 10);
    // create Shape array
    Shape arrayOfShapes[] = new Shape[3];
    // aim arrayOfShapes[0] at subclass Point object
    arrayOfShapes[0] = point;
    // aim arrayOfShapes[1] at subclass Circle object
    arrayOfShapes[1] = circle;
    // aim arrayOfShapes[2] at subclass Cylinder object
    // get name and String representation of each shape
    String output = point.getName() + "\": " + point.toString() + "\n" +
        circle.getName() + "\": " + circle.toString() + "\n" +
        cylinder.getName() + "\": " + cylinder.toString();
    // loop through arrayOfShapes and get name,
    // area and volume of each shape in arrayOfShapes
    for (int i = 0; i < arrayOfShapes.length; i++) {
        output += "\n" + arrayOfShapes[i].getName() + "\": " +
            arrayOfShapes[i].toString() + "\n" +
            arrayOfShapes[i].area() + "\n" +
            arrayOfShapes[i].volume();
    }
Shape Example: Test Class 3/3

- // get name and String representation of each shape
  - String output =
    - point.getName() + ": " + point.toString() + "\n"
    - circle.getName() + ": " + circle.toString() + "\n"
    - cylinder.getName() + ": " + cylinder.toString();
- // loop through arrayOfShapes and get name,
- // area and volume of each shape in arrayOfShapes
  - for ( int i = 0; i < arrayOfShapes.length; i++ ) {
    - output += ": " + arrayOfShapes[ i ].getName() + "\n"
      + arrayOfShapes[ i ].toString() + "\nArea = " +
      + precision2.format( arrayOfShapes[ i ].area() ) + "\nVolume = " +
      + precision2.format( arrayOfShapes[ i ].volume() );
  - }
- JOptionPane.showMessageDialog(null,output, "Demonstrating Polymorphism");
- System.exit( 0 );
- } // end class Test

Defining an Interface

```java
public interface StockWatcher {
    final String sunTicker = "SUN";
    final String oracleTicker = "ORCL";
    final String ciscoTicker = "CSCO";

    void valueChanged(String tickerSymbol, double newValue);
}
```

element

- applet that implements the StockWatcher interface:

  ```java
  public class StockApplet extends Applet implements StockWatcher {
      ...
      public void valueChanged(String tickerSymbol, double newValue) {
          if (tickerSymbol.equals(sunTicker)) {} 
          else if (tickerSymbol.equals(oracleTicker)) { ... }
          else if (tickerSymbol.equals(ciscoTicker)) { ... }
      }
  }
  ```
Warning! Interfaces Cannot Grow

• Suppose that you want to add some functionality to StockWatcher. For instance, suppose that you want to add a method that reports the current stock price, regardless of whether the value changed:
  – public interface StockWatcher {
    • final String sunTicker = "SUNW";
    • final String oracleTicker = "ORCL";
    • final String ciscoTicker = "CSCO";
    • void valueChanged(String tickerSymbol, double newValue);
    • void currentValue(String tickerSymbol, double newValue);
  – }

Warning! Interfaces Cannot Grow

• However, if you make this change, all classes that implement the old StockWatcher interface will break because
  – they don’t implement the interface anymore!