Polymorphism and Inheritance

Chapter 8
Objectives

• Describe polymorphism and inheritance in general
• Define interfaces to specify methods
• Describe dynamic binding
• Define and use derived classes in Java
• Use the class JFrame to produce windowing interfaces within Java application programs
Polymorphism: Outline

• Class Interfaces
• Java Interfaces
• Implementing an Interface
• An Interface as a Type
• Extending an Interface
Class Interfaces

• Consider a set of behaviors for pets
  ▪ Be named
  ▪ Eat
  ▪ Respond to a command

• We could specify method headings for these behaviors

• These method headings can form a class interface
Class Interfaces

• Now consider different classes that implement this interface
  ▪ They will each have the *same behaviors*
  ▪ *Nature* of the behaviors will be different

• Each of the classes implements the behaviors/methods differently
Java Interfaces

- A program component that contains headings for a number of public methods
  - Will include comments that describe the methods
- Interface can also define public named constants
- View example interface, listing 8.1
  interface Measurable
Java Interfaces

• Interface name begins with uppercase letter
• Stored in a file with suffix .java
• Interface does not include
  ▪ Declarations of constructors
  ▪ Instance variables
  ▪ Method bodies
Implementing an Interface

• To implement a method, a class must
  ▪ Include the phrase
    
    \textit{implements Interface\_name}
  ▪ Define each specified method

• View \texttt{sample class}, listing 8.2

\begin{verbatim}
class Rectangle implements Measurable
\end{verbatim}

• View another class, listing 8.3 which also implements Measurable

\begin{verbatim}
class Circle
\end{verbatim}
An Inheritance as a Type

• Possible to write a method that has a parameter as an interface type
  ▪ An interface is a reference type

• Program invokes the method passing it an object of any class which implements that interface
An Inheritance as a Type

• The method can substitute one object for another
  ▪ Called polymorphism

• This is made possible by mechanism
  ▪ Dynamic binding
  ▪ Also known as late binding
Extending an Interface

• Possible to define a new interface which builds on an existing interface
  ▪ It is said to extend the existing interface

• A class that implements the new interface must implement all the methods of both interfaces
Inheritance Basics: Outline

- Derived Classes
- Overriding Method Definitions
- Overriding Versus Overloading
- The `final` Modifier
- Private Instance Variables and Private Methods of a Base Class
- UML Inheritance Diagrams
Inheritance Basics

• Inheritance allows programmer to define a general class
• Later you define a more specific class
  ▪ Adds new details to general definition
• New class inherits all properties of initial, general class
• View example class, listing 8.4 class Person
Derived Classes

• Class **Person** used as a *base class*
  ▪ Also called *superclass*

• Now we declare *derived class* **Student**
  ▪ Also called *subclass*
  ▪ Inherits methods from the superclass

• View **derived class**, listing 8.5
  ```java
class Student extends Person
```

• View **demo program**, listing 8.6
  ```java
class InheritanceDemo
```

Sample screen output

Name: Warren Peace
Student Number: 1234
Derived Classes

- Figure 8.1 A class hierarchy
Overriding Method Definitions

• Note method `writeOutput` in class `Student`
  ▪ Class Person also has method with that name

• Method in subclass with same signature overrides method from base class
  ▪ Overriding method is the one used for objects of the derived class

• Overriding method must return same type of value
Overriding Versus Overloading

• Do not confuse overriding with overloading
  ▪ Overriding takes place in subclass – new method with same signature

• Overloading
  ▪ New method in same class with different signature
The **final** Modifier

- Possible to specify that a method **cannot** be overridden in subclass
- Add modifier final to the heading
  ```java
  public final void specialMethod()
  ```
- An entire class may be declared **final**
  - Thus cannot be used as a base class to derive any other class
Private Instance Variables, Methods

• Consider private instance variable in a base class
  ▪ It is not inherited in subclass
  ▪ It can be manipulated only by public accessor, modifier methods

• Similarly, private methods in a superclass not inherited by subclass
UML Inheritance Diagrams

- Figure 8.2 A class hierarchy in UML notation

An Employee is a Person and so forth, hence the arrows point up.
UML Inheritance Diagrams

- Figure 8.3
  Some details of UML class hierarchy from figure 8.2

```
Person
- name: String

+ setName(String newName): void
+ getName( ): String
+ writeOutput( ): void
+ hasSameName(Person otherPerson)): boolean

Student
- studentNumber: int

+ reset(String newName, int newStudentNumber): void
+ getStudentNumber( ): int
+ setStudentNumber(int newStudentNumber): void
+ writeOutput( ): void
+ equals(Student otherStudent): boolean
```
Programming with Inheritance: Outline

• Constructors in Derived Classes
• The `this` Method – Again
• Calling an Overridden Method
• Derived Class of a Derived Class
• Type Compatibility
Programming with Inheritance: Outline

• The class `Object`
• A Better `equals` Method
• Case Study: Character Graphics
• Abstract Classes
• Dynamic Binding and Inheritance
Constructors in Derived Classes

- A derived class does not inherit constructors from base class
  - Constructor in a subclass must invoke constructor from base class
- Use the reserve word `super`

```java
public Student(String initialName, int initialStudentNumber) {
    super(initialName);
    studentNumber = initialStudentNumber;
}
```

- Must be first action in the constructor
The **this** Method – Again

- Also possible to use the **this** keyword
  - Use to call any constructor in the class

```java
public Person()
{
    this("No name yet");
}
```

- When used in a constructor, this calls constructor in same class
  - Contrast use of **super** which invokes constructor of base class
Calling an Overridden Method

- Reserved word `super` can also be used to call method in overridden method

```java
public void writeOutput()
{
    super.writeOutput();  // Display the name
    System.out.println("Student Number: " + studentNumber);
}
```

- Calls method by same name in base class
Programming Example

• A derived class of a derived class
• View sample class, listing 8.7
class Undergraduate
• Has all public members of both
  ▪ Person
  ▪ Student
• This reuses the code in superclasses
Programming Example

- Figure 8.4

More details of the UML class hierarchy

<table>
<thead>
<tr>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>- studentNumber: int</td>
</tr>
<tr>
<td>+ reset(String newName, int newStudentNumber): void</td>
</tr>
<tr>
<td>+ getStudentNumber(): int</td>
</tr>
<tr>
<td>+ setStudentNumber(int newStudentNumber): void</td>
</tr>
<tr>
<td>+ writeOutput(): void</td>
</tr>
<tr>
<td>+ equals(Student otherStudent): boolean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Undergraduate</th>
</tr>
</thead>
<tbody>
<tr>
<td>- level: int</td>
</tr>
<tr>
<td>+ reset(String newName, int newStudentNumber, int newlevel): void</td>
</tr>
<tr>
<td>+ getLevel(): int</td>
</tr>
<tr>
<td>+ setLevel(int newLevel): void</td>
</tr>
<tr>
<td>+ writeOutput(): void</td>
</tr>
<tr>
<td>+ equals(Undergraduate otherUndergraduate): boolean</td>
</tr>
</tbody>
</table>
Type Compatibility

• In the class hierarchy
  ▪ Each Undergraduate is also a Student
  ▪ Each Student is also a Person

• An object of a derived class can serve as an object of the base class
  ▪ Note this is not typecasting

• An object of a class can be referenced by a variable of an ancestor type
Type Compatibility

• Be aware of the "is-a" relationship
  ▪ A **Student** is a **Person**

• Another relationship is the "has-a"
  ▪ A class can contain (as an instance variable) an object of another type
  ▪ If we specify a date of birth variable for **Person** – it "has-a" **Date** object
The Class **Object**

- Java has a class that is the ultimate ancestor of every class
  - The class **Object**
- Thus possible to write a method with parameter of type **Object**
  - Actual parameter in the call can be object of any type
- Example: method **println(Object theObject)**
The Class **Object**

- Class Object has some methods that every Java class inherits
- Examples
  - Method **equals**
  - Method **toString**
- Method **toString** called when `println(theObject)` invoked
  - Best to define your own **toString** to handle this
A Better `equals` Method

- Programmer of a class should override method `equals` from `Object`
- View code of `sample override`, listing 8.8

```java
public boolean equals (Object theObject)
```
Case Study

- Character Graphics
- View interface for simple shapes, listing 8.9 interface ShapeInterface
- If we wish to create classes that draw rectangles and triangles
  - We could create interfaces that extend ShapeInterface
  - View interfaces, listing 8.10
Case Study

• Now view base class, listing 8.11 which uses (implements) previous interfaces class ShapeBasics

• Note
  ▪ Method drawAt calls drawHere
  ▪ Derived classes must override drawHere
  ▪ Modifier extends comes before implements
Case Study

• Figure 8.5 A sample rectangle and triangle
Case Study

• Note algorithm used by method `drawHere` to draw a rectangle
  1. Draw the top line
  2. Draw the side lines
  3. Draw the bottom lines

• Subtasks of `drawHere` are realized as private methods

• View class definition, listing 8.12

```
class Rectangle
```
Case Study

• View **next class** to be defined (and tested), listing 8.13  class Triangle

• It is a good practice to test the classes as we go

• View **demo program**, listing 8.14  class TreeDemo
Case Study

Save the Redwoods!

* *
* *
* *
* *
* *
* *
* *
* *
* *
* *

**********************

Sample screen output
Abstract Classes

• Class **ShapeBasics** is designed to be a base class for other classes
  ▪ Method **drawHere** will be redefined for each subclass
  ▪ It should be declared **abstract** – a method that has no body
• This makes the **class** abstract
• You cannot create an object of an abstract class – thus its role as base class
Abstract Classes

• Not all methods of an abstract class are abstract methods

• Abstract class makes it easier to define a base class
  ▪ Specifies the obligation of designer to override the abstract methods for each subclass
Abstract Classes

- Cannot have an instance of an abstract class
  - But OK to have a parameter of that type
- View **abstract version**, listing 8.15
  ```java
  abstract class ShapeBase
  ```
Dynamic Binding and Inheritance

• Note how `drawAt` (in `ShapeBasics`) makes a call to `drawHere`

• Class `Rectangle` overrides method `drawHere`
  ▪ How does `drawAt` know where to find the correct `drawHere`?

• Happens with dynamic or late binding
  ▪ Address of correct code to be executed determined at run time
Dynamic Binding and Inheritance

• When an overridden method invoked
  ▪ Action matches method defined in class used to create object using `new`
  ▪ Not determined by type of variable naming the object

• Variable of any ancestor class can reference object of descendant class
  ▪ Object always remembers which method actions to use for each method name
Graphics Supplement: Outline

- The Class `JApplet`
- The Class `JFrame`
- Window Events and Window Listeners
- The `ActionListener` Interface
- Programming Example: `HappyFace` as a `JFrame` GUI
The Class **JApplet**

- Class **JApplet** is a base class for all applets
  - Has methods `init` and `paint`
- When you extend **JApplet** you override (redefine) these methods
- Parameter shown will use your versions due to polymorphism

```java
public void showApplet(JApplet anApplet) {
    anApplet.init();
    ...
    anApplet.paint();
}
```
The Class `JFrame`

- For GUIs to run as applications (instead of from a web page)
  - Use class `JFrame` as the base class
- View example program, listing 8.16 class `ButtonDemo`
- Note method `setSize`
  - Width and height given in number of pixels
  - Sets size of window
The Class `JFrame`

- View demo program, listing 8.17 class `ShowButtonDemo`
Window Events and Window Listeners

• Close-window button fires an event
  - Generates a window event handled by a window listener

• View class for window events, listing 8.18, class WindowDestroyer

• Be careful not to confuse JButtons and the close-window button
The **ActionListener** Interface

- Use of interface ActionListener requires only one method
  ```java
  public void actionPerformed(ActionEvent e)
  ```
- Listener that responds to button clicks
  - Must be an action listener
  - Thus must **implement** ActionListener interface
Programming Example

- **HappyFace** as a **JFrame** GUI
- View **class** with **JFrame** window, listing 8.19, **class HappyFace**
- Note **demo program**, listing 8.20 **class ShowHappyFace**
Summary

• An interface contains
  ▪ Headings of public methods
  ▪ Definitions of named constants
  ▪ No constructors, no private instance variables

• Class which implements an interface must
  ▪ Define a body for every interface method specified

• Interface enables designer to specify methods for another programmer
Summary

• Interface is a reference type
  ▪ Can be used as variable or parameter type
• Interface can be extended to create another interface
• Dynamic (late) binding enables objects of different classes to substitute for one another
  ▪ Must have identical interfaces
  ▪ Called polymorphism
Summary

• Derived class obtained from base class by adding instance variables and methods
  ▪ Derived class inherits all public elements of base class
• Constructor of derived class must first call a constructor of base class
  ▪ If not explicitly called, Java automatically calls default constructor
Summary

- Within constructor
  - `this` calls constructor of same class
  - `super` invokes constructor of base class
- Method from base class can be overridden
  - Must have same signature
- If signature is different, method is overloaded
Summary

• Overridden method can be called with preface of `super`

• Private elements of base class cannot be accessed directly by name in derived class

• Object of derived class has type of both base and derived classes

• Legal to assign object of derived class to variable of any ancestor type
Summary

- Every class is descendant of class `Object`
- Class derived from `JFrame` produces applet like window in application program
- Method `setSize` resizes `JFrame` window
- Class derived from `WindowAdapter` defined to be able to respond to `closeWindow` button