INHERITANCE, POLYMORPHISM, AND INTERFACES

CODE EXAMPLES FROM JAVA: AN INTRODUCTION TO PROGRAMMING AND PROBLEM SOLVING (6TH EDITION), BY WALTER SAVITCH

IQS2: Spring 2013
Objectives

- Describe polymorphism and inheritance in general
- Define interfaces to specify methods
- Describe dynamic binding
- Define and use derived classes in Java
- Understand how inheritance is used in the JFrame class
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
- Example: the Person class
LISTING 8.1  The Class Person

```java
public class Person
{
    private String name;

    public Person()
    {
        name = "No name yet";
    }

    public Person(String initialName)
    {
        name = initialName;
    }

    public void setName(String newName)
    {
        name = newName;
    }

    public String getName()
    {
        return name;
    }

    public void writeOutput()
    {
        System.out.println("Name: " + name);
    }

    public boolean hasSameName(Person otherPerson)
    {
        return this.name.equalsIgnoreCase(otherPerson.name);
    }
}
```
Derived Classes

- An example class hierarchy
Derived Classes

- **Person** class used as a *base* class
  - Also called *superclass*

- **Student** is a *derived* class
  - Also called *subclass*
  - Inherits methods and members from the superclass
public class Student extends Person {
    private int studentNumber;
    public Student() {
        super();
        studentNumber = 0; // Indicating no number yet
    }
    public Student(String initialName, int initialStudentNumber) {
        super(initialName);
        studentNumber = initialNumber;
    }
    public void reset(String newName, int newStudentNumber) {
        setName(newName);
        studentNumber = newStudentNumber;
    }
    public int getStudentNumber() {
        return studentNumber;
    }
    public void setStudentNumber(int newStudentNumber) {
        studentNumber = newStudentNumber;
    }
    public void writeOutput() {
        System.out.println("Name: " + getName());
        System.out.println("Student Number: " + studentNumber);
    }
}
public boolean equals(Student otherStudent)
{
    return this.hasSameName(otherStudent) &&
    (this.studentNumber == otherStudent.studentNumber);
}
**LISTING 8.3**  A Demonstration of Inheritance Using Student

```java
public class InheritanceDemo {
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("Warren Peace"); // setName is inherited from the class Person.
        s.setStudentNumber(1234);
        s.writeOutput();
    }
}
```

**Screen Output**

```
Name: Warren Peace
Student Number: 1234
```
Don’t Recode What Is Already Coded!

- When you implement a subclass, you get:
  - All of the data members of the base class…
    - …though you may not be able to access them the way you’d like. More on that later.
  - All of the methods of the base class…
    - …with same caveat as above

- So don’t add or recode them in the subclass!
- BUT, it may be that you don’t like the way some methods are coded/used in the base class. In that case…
Overriding Method Definitions

- Note method `writeOutput` in class `Student`.
  - Class `Person` also has a method with that name.
- Method in subclass with `same signature overrides` method from base class.
  - When an instance of the `Student` class calls the `writeOutput()` method, the version of the method that is run is the one shown in the `Student` 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
    - Example: In `String` class:

```
int indexOf(String str)
   Returns the index within this string of the first occurrence of the specified substring.

int indexOf(String str, int fromIndex)
   Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
```
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

- Included here for completeness: I’ve never used the **final** modifier for an entire class.
Private Instance Variables, Methods

- **private** instance variable in a base class
  - Are inherited in subclass (despite what your text may say), but can’t be directly manipulated by you
  - Can only be manipulated by public accessor, modifier methods

- Similarly, **private** methods in a superclass cannot be called in your subclass code
  - Which at times, is not pleasant. But it’s almost always OK. Why?
Protected Instance Variables, Methods

- protected instance variables and methods in a base class
  - Can be used any way you want in any descendent class of the base class
  - Can be used any way you want inside any method in any class in the same package
    - See Appendix 5 in your text
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
  - Again, here for completeness
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
A derived class of a derived class: **Undergraduate** class

Has all public members of both

- **Person**
- **Student**

This reuses the code in superclasses
LISTING 8.4  A Derived Class of a Derived Class

```java
public class Undergraduate extends Student {
    private int level; // 1 for freshman, 2 for sophomore
                          // 3 for junior, or 4 for senior.
    public Undergraduate() {
        super();
        level = 1;
    }
    public Undergraduate(String initialName,
                          int initialStudentNumber, int initialLevel) {
        super(initialName, initialStudentNumber);
        setLevel(initialLevel); // checks 1 <= initialLevel <= 4
    }
    public void reset(String newName, int newStudentNumber,
                       int newLevel) {
        set(newName, newStudentNumber); // Student's reset
        setLevel(newLevel); // Checks 1 <= newLevel <= 4
    }
}
```
public int getLevel()
{
    return level;
}

public void setLevel(int newLevel)
{
    if ((1 <= newLevel) && (newLevel <= 4))
        level = newLevel;
    else
    {
        System.out.println("Illegal level!");
        System.exit(0);
    }
}

public void writeOutput()
{
    super.writeOutput();
    System.out.println("StudentLevel: " + level);
}

public boolean equals(Undergraduate otherUndergraduate)
{
    return equals(Student)otherUndergraduate) &&
        (this.level == otherUndergraduate.level);
}
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 (that is, used wherever the base class is required)
  - Ex: as input parameters to methods
  - Note this is not typecasting

- An object of a class can be referenced by a variable of an ancestor type
  - So, for example, a **Person** variable can point to (reference) an **Undergraduate** object (but not vice versa)
Type Compatibility

- Be aware of the "is-a" relationship
  - An Undergraduate is a Person
  - But a Person is not necessarily an Undergraduate

- 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)`
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 a better `equals` method
  ```java
  public boolean equals (Object theObject)
  ```
LISTING 8.5  A Better equals Method for the Class Student

```java
public boolean equals(Object otherObject)
{
    boolean isEqual = false;
    if ((otherObject != null) &&
        (otherObject instanceof Student))
    {
        Student otherStudent = (Student)otherObject;
        isEqual = this.sameName(otherStudent) &&
            (this.studentNumber ==
                otherStudent.studentNumber);
    }
    return isEqual;
}
```
Polymorphism

- Inheritance allows you to define a base class and derive classes from the base class.
- Polymorphism allows you to make changes in the method definition for the derived classes and have those changes apply to methods written in the base class.
Polymorphism

Consider an array of **Person**

```java
Person[] people = new Person[4];
```

Since **Student** and **Undergraduate** are types of **Person**, we can assign them to **Person** variables

```java
people[0] = new Student("DeBanque, Robin", 8812);
people[1] = new Undergraduate("Cotty, Manny", 8812, 1);
```
Polymorphism

- Given:
  ```java
  Person[] people = new Person[4];
  people[0] = new Student("DeBanque, Robin", 8812);
  ```
- When invoking:
  ```java
  people[0].writeOutput();
  ```
- Which `writeOutput()` is invoked, the one defined for `Student` or the one defined for `Person`?
- Answer: The one defined for `Student`
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*
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
Polymorphism Example

- View sample class, listing 8.6

```java
class PolymorphismDemo
```

- Output

```
Name: Cotty, Manny
Student Number: 4910
Student Level: 1

Name: Kick, Anita
Student Number: 9931
Student Level: 2

Name: DeBanque, Robin
Student Number: 8812

Name: Bugg, June
Student Number: 9901
Student Level: 4
```
LISTING 8.6 A Demo of Polymorphism (part 1 of 2)

public class PolymorphismDemo
{
    public static void main(String[] args)
    {
        Person[] people = new Person[4];
        people[0] = new Undergraduate("Cotty, Manny", 4910, 1);
        people[1] = new Undergraduate("Kick, Anita", 9931, 2);
        people[2] = new Student("DeBanque, Robin", 8812);
        people[3] = new Undergraduate("Bugg, June", 9901, 4);
        for (Person p : people)
        {
            p.writeOutput();
            System.out.println();
        }
    }
}

Even though p is of type Person, the writeOutput method associated with Undergraduate or Student is invoked depending upon which class was used to create the object.

Screen Output

Name: Cotty, Manny  
Student Number: 4910  
Student Level: 1  

Name: Kick, Anita  
Student Number: 9931  
Student Level: 2
LISTING 8.6 A Demo of Polymorphism (part 2 of 2)

<table>
<thead>
<tr>
<th>Name</th>
<th>Student Number</th>
<th>Name</th>
<th>Student Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeBanque, Robin</td>
<td>8812</td>
<td>Bugg, June</td>
<td>9901</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Student Level</td>
<td>4</td>
<td>Student Level</td>
<td>4</td>
</tr>
</tbody>
</table>
An Aside: Types and Security

- Java is a “strongly typed” language
  - This means that it is very careful about making sure appropriate typed objects are passed to methods and assigned as references.

- All other factors being equal, strong typing makes a language much more secure.
  - Can anyone guess why this is?

- But, it turns out that the Java type system can be fooled via careful (mis)use of the dynamic binding system!
  - And if you manage to fool it even once, you have rendered the type system completely ineffective!
  - The method researchers discovered for doing this is considered so dangerous that it has never been published!
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
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
LISTING 8.7  A Java Interface

/**
 * An interface for methods that return the perimeter and area of an object.
 */
public interface Measurable
{
    /** Returns the perimeter. */
    public double getPerimeter();
    /** Returns the area. */
    public double getArea();
}
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 `implements Interface_name`
  - Define each specified method
LISTING 8.8  An Implementation of the Interface Measurable

A class of rectangles.

```java
/**
 * A class of rectangles.
 */
public class Rectangle implements Measurable {
    private double myWidth;
    private double myHeight;

    public Rectangle(double width, double height)
    {
        myWidth = width;
        myHeight = height;
    }

    public double getPerimeter()
    {
        return 2 * (myWidth + myHeight);
    }

    public double getArea()
    {
        return myWidth * myHeight;
    }
}
```
LISTING 8.9  Another Implementation of the Interface  
Measurable

```java
/**
 * A class of circles.
 */
public class Circle implements Measurable
{
    private double myRadius;
    public Circle(double radius)
    {
        myRadius = radius;
    }
    public double getPerimeter()
    {
        return 2 * Math.PI * myRadius;
    }
    public double getCircumference()
    {
        return getPerimeter();
    }
    public double getArea()
    {
        return Math.PI * myRadius * myRadius;
    }
}
```

This method is not declared in the interface.

Calls another method instead of repeating its body.
An Inheritance as a Type

- Possible to write a method that has an Interface type as a parameter
  - An interface is a reference type
- Program invokes the method, passing it an object of any class which implements that interface
Example: Genetic Algorithm

- A Population described by chromosomes
- Crossover
- Mutation
- Survival of the fittest
  - Fitness function
Flow Diagram of the Genetic Algorithm Process

1. Describe Problem
2. Generate Initial Solutions
3. Test: is initial solution good enough?
   - Yes: Stop
   - No: Select parents to reproduce
4. Apply crossover process and create a set of offspring
5. Apply random mutation
Figure 2. Photographs of prototype evolved antennas: (a) the best evolved antenna for the initial gain pattern requirement, ST5-3-10; (b) the best evolved antenna for the revised specifications, ST5-33-142-7.
The Comparator Interface

- Required for use in Java Arrays class
  - `Arrays.sort()`
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
Java has many predefined interfaces

One of them, the **Comparable** interface, is used to impose an ordering upon the objects that implement it

Requires that the method `compareTo` be written

```java
public int compareTo(Object other);
```
Sorting an Array of Fruit Objects

- Initial (non-working) attempt to sort an array of Fruit objects
- View class definition, listing 8.16
class Fruit
- View test class, listing 8.17
class FruitDemo
- Result: Exception in thread “main”
  - Sort tries to invoke compareTo method but it doesn’t exist
Sorting an Array of Fruit Objects

- Working attempt to sort an array of Fruit objects – implement Comparable, write compareTo method
- Following slides show Fruit class
- Result: Exception in thread “main”
  - Sort tries to invoke method but it doesn’t exist
LISTING 8.16  First Attempt to Define a Fruit Class

public class Fruit
{
    private String fruitName;
    public Fruit()
    {
        fruitName = "";
    }
    public Fruit(String name)
    {
        fruitName = name;
    }
    public void setName(String name)
    {
        fruitName = name;
    }
    public String getName()
    {
        return fruitName;
    }
}
Listing 8.17  Program to Sort an Array of Fruit Objects

```java
import java.util.Arrays;
public class FruitDemo
{
    public static void main(String[] args)
    {
        Fruit[] fruits = new Fruit[4];
        fruits[0] = new Fruit("Orange");
        fruits[1] = new Fruit("Apple");
        fruits[2] = new Fruit("Kiwi");
        fruits[3] = new Fruit("Durian");
        Arrays.sort(fruits);
        // Output the sorted array of fruits
        for (Fruit f : fruits)
        {
            System.out.println(f.getName());
        }
    }
}
```
LISTING 8.18  A Fruit Class implementing Comparable
(part 1 of 2)

```java
public class Fruit implements Comparable {
    private String fruitName;
    public Fruit() {
        fruitName = "";
    }
    public Fruit(String name) {
        fruitName = name;
    }
}
```
public void setName(String name) {
    fruitName = name;
}

public String getName() {
    return fruitName;
}

public int compareTo(Object o) {
    if (((o != null) &&
        (o instanceof Fruit))
    {
        Fruit otherFruit = (Fruit) o;
        return (fruitName.compareTo(otherFruit.fruitName));
    }
    return -1;  // Default if other object is not a Fruit
An alternate definition that will sort by length of the fruit name

```java
public int compareTo(Object o)
{
    if ((o != null) &&
        (o instanceof Fruit))
    {
        Fruit otherFruit = (Fruit) o;
        if (fruitName.length() >
            otherFruit.fruitName.length())
            return 1;
        else if (fruitName.length() <
            otherFruit.fruitName.length())
            return -1;
        else
            return 0;
    }
    return -1; // Default if other object is not a Fruit
}
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
How does Java know which version of a method is to be run?

Happens with dynamic or late binding
  - Address of correct code to be executed determined at run time
Graphics Supplement: Outline

- The Class JApplet
- The Class JFrame
- Window Events and Window Listeners
- The ActionListener Interface
The Class `JApplet`

- Class `JApplet` is 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 8.20
  - class **ButtonDemo**
- Note method **setSize**
  - Width and height given in number of pixels
  - Sets size of window
The Class JFrame

- View demo program, listing 8.21 class ShowButtonDemo

Sample screen output
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 8.22, 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
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