Inheritance
Introduction

Inheritance is an important mechanism for developing reusable, robust software components. Through the use of inheritance, programmers can reuse the code in the superclass many times.

Objectives

- To understand the concepts of inheritance
- To understand how to inherit and override superclass methods
- To be able to invoke superclass constructors
- To understand the common superclass Object and to override its toString method
- To use inheritance for customizing user interfaces

Inheritance

Object-oriented systems allow classes to be defined in terms of other classes. For example, mountain bikes, road bikes, and tandems are all kinds of bicycles. In object-oriented terminology, mountain bikes, road bikes, and tandems are all subclasses of the bicycle class. Similarly, the bicycle class is the superclass of mountain bikes, road bikes, and tandems.

Each subclass inherits state and behaviours from the superclass. However, subclasses are not limited to the state and behaviours provided to them by their superclass. Subclasses can add variables and methods to the ones they inherit from the superclass. For example, tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Subclasses can also override inherited methods and provide specialized implementations for those methods. For example, if you had a mountain bike with an additional chain ring, you would override the "change gears" method so that the rider could shift into those lower gears.

You are not limited to just one layer of inheritance. The inheritance tree, or class hierarchy, can be as deep as needed. Methods and variables are inherited down through the levels. In general, the farther down in the hierarchy a class appears, the more specialized its behaviour is.

The Object class, defined in the java.lang package, defines and implements behaviour that every class needs. The Object class is at the top of the class hierarchy, every Java class extends the object class either directly or indirectly.
Inheriting Instance Fields and Methods

Additional instance fields and methods can be specified when defining a subclass of a given class. The following considerations need to be taken when defining methods and fields for a subclass.

Methods: There are three possibilities when defining the methods for a subclass:

1. Override methods from the superclass. If a method in the subclass has the same signature (same method name and the parameter types) as a method in its superclass, it overrides the method of the superclass.
2. Inherit methods from the superclass. The superclass method can be applied to the subclass objects.
3. Define new methods. If you define a method that did not exist in the superclass, then the new method can be applied only to subclass objects.

Fields: There are two cases when defining the instance fields for a subclass.

1. The subclass inherits all fields from the superclass.
2. Any new instance fields that you define in the subclass are present only in subclass objects.

Please note, a subclass has no access to private fields of its superclass. If you want to modify a private superclass field, you must use a public method of the superclass. To call a method of the superclass instead of the method of the current class, a keyword super is used. Chapter 13.3 has details.

Subclass Construction

To call the superclass constructor, use the super keyword followed by parenthesis, it indicates a call to the superclass constructor. When used in this way, the constructor call must be in the first statement of the subclass constructor.

As you can see, a super keyword can be used in calling superclasses’ constructors as well as superclasses’ methods. In the following example, a CheckingAccount class inherits BankAccount class, CheckingAccount class calls the constructor from the BankAccount and overrides the methods of the BankAccount.
//superclass
public class BankAccount
{
    private double balance;  //private superclass field

    public BankAccount(double initialBalance) // superclass constructor
    {
        balance = initialBalance;
    }

    public void deposit(double amount)
    {
        balance = balance + amount;
    }

    public void withdraw(double amount)
    {
        balance = balance – amount;
    }

    public double getBalance()
    {
        return balance;
    }
}

//subclass
public class CheckingAccount extends BankAccount
{
    private int transactionCount;   //subclass field

    public CheckingAccount(double initialBalance) //subclass constructor
    {
        super(initialBalance);  //call superclass constructor in the first statement
        transactionCount = 0;
    }

    public void withdraw(double amount) //override withdraw method in superclass
    {
        transactionCount++;
        /* next line has error
         balance = balance – amount;   // cannot access private field of superclass */
        super.withdraw(amount);  // call superclass method
    }

    ......
Converting Between Subclass and Superclass Types

Subclass references can be converted to superclass references. For example,

    CheckingAccount harrysAccount = new CheckingAccount(1000);
    BankAccount anAccount = harrysAccount;

With this conversion, the object reference anAccount knows less than the object reference harrysAccount because BankAccount is the superclass of CheckingAccount. Why would anyone want to know less about an object and store a reference in an object field of a superclass? This can happen if you want to reuse code that knows about the superclass but not the subclass.

Access Control

Java has four levels of controlling access to fields, methods and classes:

- public access – accessible by all classes of all packages
- private access – accessible only by the methods of the class
- protected access – accessible by subclasses
- package access – (default, no access modifier is given)

Class Object – An Ancestor of All Java Classes

Object class is the ancestor of all Java classes. The most common used methods defined in object class are

- String toString( ) – Returns a string representation of the object
- boolean equals(Object otherObject) – Tests whether the object equals another object
- Object clone( ) – Makes a full copy of an object

Self reading: Chapter 13.8.1 Overriding the toString method, page 496

Processing Text Input

Except for the showInputDialog method of the JOptionPane class, most GUI’s collect text input through text fields. The JTextField class (in javax.swing) provides a text field. For example:

    final int FIELD_WIDTH = 10;
    final JTextField rateField = new JTextField(FIELD_WIDTH);

To each text field, you generally need to provide a label so that the user knows what to type into it. Also, you should supply a button that the user can press to indicate that the input is ready for processing.

When the button is clicked, its actionPerformed method reads the user input from the text fields, using the getText method of the JTextField class. Note the getText method returns a String object.
An example is given below, pay particular attention to the actionPerformed method.

```java
import java.awt.event.*;
import javax.swing.*;

public class TestTextField
{
  public static void main(String[] args)
  {
    JFrame frame = new JFrame();
    JButton button = new JButton("Submit");
    final JLabel label = new JLabel("Hello");
    JPanel panel = new JPanel(); //to create a JPanel object
    final JTextField inputName = new JTextField(10);
    JLabel label1 = new JLabel("Input name:");
    panel.add(label1);
    panel.add(inputName);
    panel.add(button);
    panel.add(label);
    frame.add(panel);

    class AddListener implements ActionListener
    {
      public void actionPerformed(ActionEvent e)
      {
        label.setText("Hello " + inputName.getText());
      }
    }
    ActionListener listener = new AddListener();
    button.addActionListener(listener);
    frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
    frame.setVisible(true);
  }

  private static final int FRAME_WIDTH=400;
  private static final int FRAME_HEIGHT=100;
}
```

**Text Areas**

Use a JTextArea to show multiple lines of text. For example,

```java
final int ROWS = 10;
final int COLUMNS = 30;
JTextArea t = new JTextArea(ROWS, COLUMNS);
```

The common used methods for a JTextArea are: setText, append, setEditable etc. You may need to inspect the methods in the API reference when needed.
Reading

Text book:
  Chapter 10

Review questions

Review exercises:

Programming exercises:
  Page 512: Exercise P10.4, Exercise P10.9

References

Cay Horstmann, Big Java
Lab session

1) Exercise P10.4, complete the implementation to the follow 3 programs Person, Student and Instructor. Finally, write a program to create few objects of Person, Students and Instructors and test the programs.

Person.java

```java
/**
 * A person is represented by the name and a birth year.
 */
public class Person {

    /**
     * Construct a Person object.
     * @param n the name of the person
     * @param byear the birth year
     */
    public Person(String n, int byear)
    {
        // add code here to construct a Person object
    }

    /**
     * Returns the string representation of the object.
     * @return a string representation of the object
     */
    public String toString()
    {
        // add code here
    }

    private String name;
    private int birthYear;
}
```
public class Student extends Person
{
    public Student(String n, int byear, String m)
    {
        // call the super class constructor
        // add extra code to construct a Student object
    }
    public String toString()
    {
        //add implementation here
    }
    private String major;
}
Instructor.java

/**
 * An instructor is represented by a name, a birth year, and a salary.
 */
public class Instructor extends Person
{
    /**
     * Construct an Instructor object.
     * @param n the name of the instructor
     * @param byear the birth year
     * @param s the salary
     */
    public Instructor(String n, int byear, double s)
    {
        //call super class constructor
        //add extra code for Instructor
    }

    /**
     * Returns the string representation of the object.
     * @return a string representation of the object
     */
    public String toString()
    {
        //implementation
    }

    private double salary;
2) Based on the follow code examples, what will be the output when the AreaTest (next page) program is executed?

```java
public class Figure {
    double dim1;
    double dim2;
    Figure (double a, double b) {
        dim1 = a;
        dim2 = b;
    }
    public double area() {
        System.out.println("Inside area for Figure.");
        return -1;
    }
}

public class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a,b);
    }
    public double area() {
        System.out.println("Inside area for Rectangle.");
        return dim1 *dim2;
    }
}

public class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a,b);
    }
    public double area() {
        System.out.println("Inside area for Triangle.");
        return dim1 *dim2 / 2;
    }
}
```
public class AreaTest
{
    public static void main(String[] args)
    {
        Rectangle r = new Rectangle(9,5);
        Triangle t = new Triangle(10,8);
        Figure f;
        f = r;
        System.out.println("Area is " + f.area());
        f = t;
        System.out.println("Area is " + f.area());
        Figure f1 = new Figure(3,23);
        System.out.println("Area is " + f1.area());
        Figure f2 = new Rectangle(3,23);
        System.out.println("Area is " + f2.area());
        Figure f3 = new Triangle(3,23);
        System.out.println("Area is " + f3.area());
    }
}
3) Implement a digital clock by using Timer and Calendar class defined by java.util library.
Hints:
   • You need to create a frame which contain a text field to display time
   • You need to have a Timer object which has an ActionListener object attached

   e.g.

   Timer t = new Timer (1000, new ClockListener());
   t.start();

   • To use Calendar class to get time like this

   Calendar now = Calendar.getInstance();
   int h = now.get(Calendar.HOUR_OF_DAY);
   int m = now.get(Calendar.MINUTE);
   int s = now.get(Calendar.SECOND);

4) Exercise P10.9, this exercise focuses on how to implement a GUI application by extending
   JFrame class. This example separates the main method from the frame class, and use inner
   classes for the action listeners. It also shows how to use helper methods to make the program
   easy to maintain. Inspect these programs and make sure you understand them.

BankAccountViewer.java

import javax.swing.JFrame;
/**
   * A GUI for manipulating a bank account.
   */
public class BankAccountViewer {
  public static void main(String[] args) {
    BankAccount account = new BankAccount(INITIAL_BALANCE);
    // construct the frame
    JFrame frame = new BankAccountFrame(account);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
  private static final double INITIAL_BALANCE = 1000;
}
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
 * A frame for manipulating a bank account.
 */
public class BankAccountFrame extends JFrame {

    /**
     * Constructs a BankAccountFrame for a given account
     * @param anAccount the account to manipulate
     */
    public BankAccountFrame(BankAccount anAccount) {
        account = anAccount;

        // The label for displaying the results
        label = new JLabel("balance=" + account.getBalance());

        // the label, text field and button for entering an amount
        JLabel amountLabel = new JLabel("Amount:");
        amountField = new JTextField(7);

        // the control panel that holds the components
        JPanel controlPanel = new JPanel();
        controlPanel.add(amountLabel);
        controlPanel.add(amountField);
        controlPanel.add(createWithdrawButton());
        controlPanel.add(createDepositButton());
        controlPanel.add(label);

        add(controlPanel);
        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }

    private JButton createDepositButton() {
        JButton depositButton = new JButton("Deposit");

        class DepositListener implements ActionListener {
            public void actionPerformed(ActionEvent event) {
                double depositAmount = Double.parseDouble(amountField.getText());
                account.deposit(depositAmount);
                label.setText("balance=" + account.getBalance());
            }
        }

        ActionListener listener1 = new DepositListener();
        depositButton.addActionListener(listener1);
        return depositButton;
    }
}
private JButton createWithdrawButton()
{
    JButton withdrawButton = new JButton("Withdraw");

    class WithdrawListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            double withdrawAmount = Double.parseDouble(amountField.getText());
            account.withdraw(withdrawAmount);
            label.setText("balance=" + account.getBalance());
        }
    }

    ActionListener listener2 = new WithdrawListener();
    withdrawButton.addActionListener(listener2);
    return withdrawButton;
}

private BankAccount account;
private JLabel label;
private JTextField amountField;

private static final int FRAME_WIDTH = 400;
private static final int FRAME_HEIGHT = 100;

BankAccount.java

/**
 * A bank account has a balance that can be changed by deposits and withdrawals.
 */
public class
{
    /**
     * Constructs a bank account with a zero balance.
     */
    public BankAccount()
    {
        balance = 0;
    }

    /**
     * Constructs a bank account with a given balance.
     * @param initialBalance the initial balance
     */
    public BankAccount(double initialBalance)
    {
        balance = initialBalance;
    }

    /**
     * Deposits money into the bank account.
     * @param amount the amount to deposit
     */
    public void deposit(double amount)
    {
double newBalance = balance + amount;
balance = newBalance;
}

/**
 * Withdraws money from the bank account.
 * @param amount the amount to withdraw
 */
public void withdraw(double amount)
{
    double newBalance = balance - amount;
balance = newBalance;
}

/**
 * Gets the current balance of the bank account.
 * @return the current balance
 */
public double getBalance()
{
    return balance;
}

private double balance;