Design Classes and 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. This module talks about inheritance and how to design better quality classes.

Objectives

- To learn how to choose appropriate classes to implement
- To understand static methods, static field and packages
- 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 learn how to implement helper classes as inner classes
- To understand how inner classes access variables from the surrounding scope

Object, Classes, static fields, static methods and Packages

Most useful programs don’t just manipulate simple data types like numbers and strings. Instead, they deal with data items that are more complex and that more closely represent entities in the real world. Examples like bank accounts, employee records, student records. Such data items contain a set of data with complex types. For example, a student record may contain; student ID, name, address, program undertaken, course information. In Object-Oriented programming, such data items are modelled as Objects. Classes are defined to describe the behaviour of these objects.

Any Java application consists of one or more Java classes. A Java class defines the variables in its data fields and methods to manipulate objects. A constructor method is a special method in a class to offer construct objects. Every stand-alone Java application must have a main() method. The main() method can be used for testing a class by creating objects of the class and making the method calls. It can be included in either the same class or in another individual class.

Static fields and Static methods belong to entire class objects. They are not invoked on an object, i.e., when you use these fields and methods, you do not need to specify object reference, just call them by using

ClassName.someFiled  or  
ClassName.someMethod

For example,

Math.PI  (PI is a static field defined in Math class)
Math.sqrt(Math.PI)  (class static sin( ) method defined by Math class)

**Packages** organize related classes into groups. Details see Chapter 9, section 9.9.

**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.

The syntax for inheritance can be found on page 471.

Please examine the BankAccount example in Chapter 13.1 and read through Chapter 13.2 about Inheritance Hierarchies for further understanding.

**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.

```java
//superclass
public class BankAccount
{
    private double balance; //private superclass field

    public BankAccount(double intialBalance) // superclass constructor
    {
        balance = intialBalance;
    }
    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 intialBalance) //subclass constructor
    {
        super(intialBalance); //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. The detailed description is in Chapter 13.5 on page 482 to 484. You should read through this section for further understanding.

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

**Reading**

Text book:
- Chapter 9: 9.1, 9.6, 9.7, and 9.9
- Chapter 13: whole chapter

**Review questions**

Review exercises:

Programming exercises:
- Page 512: Exercise P13.4

**References**

Solution to Programming Exercise

P13-4
/**
 * This class tests the Person, Student, and Instructor classes.
 */
public class ExP13_4
{
    public static void main(String[] args)
    {
        Person p = new Person("Perry", 1959);
        Student s = new Student("Sylvia", 1979, "Computer Science");
        Instructor e = new Instructor("Edgar", 1969, 65000);
        System.out.println(p);
        System.out.println(s);
        System.out.println(e);
    }
}

/**
 * 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)
     { super(n, byear); salary = s; }
     */

    /**
     * Returns the string representation of the object.
     * @return a string representation of the object
     *
     * public String toString()
     { return "Instructor[super=" + super.toString() + ",salary=" + salary + "]"; }
     */

    private double salary;
}

/**
 * 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)
    {
        name = n;
        birthYear = byear;
    }

    /**
     * Returns the string representation of the object.
     * @return a string representation of the object
     */
    public String toString()
    {
        return "Person[name=" + name + ", birthYear=" + birthYear + "]";
    }

    private String name;
    private int birthYear;
}

/**
 * A student is represented by the name, birth year, and major.
 */
public class Student extends Person
{
    /**
     * Construct a Student object.
     * @param n the name of the student
     * @param byear the birth year
     * @param m the major
     */
    public Student(String n, int byear, String m)
    {
        super(n, byear);
        major = m;
    }

    /**
     * Returns the string representation of the object.
     * @return a string representation of the object
     */
    public String toString()
    {
        return "Student[super=" + super.toString() + ", major=" + major + "]";
    }

    private String major;
}