Chapter 10 Thinking in Objects
Motivations

You see the advantages of object-oriented programming from the preceding two chapters. This chapter will demonstrate how to solve problems using the object-oriented paradigm. Before studying these examples, we first introduce several language features for supporting these examples.
Objectives

- To create immutable objects from immutable classes to protect the contents of objects (§10.2).
- To determine the scope of variables in the context of a class (§10.3).
- To use the keyword `this` to refer to the calling object itself (§10.4).
- To apply class abstraction to develop software (§10.5).
- To explore the differences between the procedural paradigm and object-oriented paradigm (§10.6).
- To develop classes for modeling composition relationships (§10.7).
- To design programs using the object-oriented paradigm (§§10.8-10.10).
- To design classes that follow the class-design guidelines (§10.11).
Immutable Objects and Classes

If the contents of an object cannot be changed once the object is created, the object is called an *immutable object* and its class is called an *immutable class*. If you delete the set method in the `Circle` class in the preceding example, the class would be immutable because radius is private and cannot be changed without a set method.

A class with all private data fields and without mutators is not necessarily immutable. For example, the following class `Student` has all private data fields and no mutators, but it is mutable.
public class Student {
    private int id;
    private BirthDate birthDate;

    public Student(int ssn, int year, int month, int day) {
        id = ssn;
        birthDate = new BirthDate(year, month, day);
    }

    public int getId() {
        return id;
    }

    public BirthDate getBirthDate() {
        return birthDate;
    }
}

public class BirthDate {
    private int year;
    private int month;
    private int day;

    public BirthDate(int newYear, int newMonth, int newDay) {
        year = newYear;
        month = newMonth;
        day = newDay;
    }

    public void setYear(int newYear) {
        year = newYear;
    }

    public void setMonth(int newMonth) {
        month = newMonth;
    }

    public void setDay(int newDay) {
        day = newDay;
    }
}

public class Test {
    public static void main(String[] args) {
        Student student = new Student(111223333, 1970, 5, 3);
        BirthDate date = student.getBirthDate();
        date.setYear(2010); // Now the student birth year is changed!
    }
}
What Class is Immutable?

For a class to be immutable, it must mark all data fields private and provide no mutator methods and no accessor methods that would return a reference to a mutable data field object.
Scope of Variables

- The scope of instance and static variables is the entire class. They can be declared anywhere inside a class.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be initialized explicitly before it can be used.
The this Keyword

- The **this** keyword is the name of a reference that refers to an object itself. One common use of the **this** keyword is to reference a class’s *hidden data fields*.
- Another common use of the **this** keyword to enable a constructor to invoke another constructor of the same class.
Reference the Hidden Data Fields

```java
public class Foo {
    private int i = 5;
    private static double k = 0;

    void setI(int i) {
        this.i = i;
    }

    static void setK(double k) {
        Foo.k = k;
    }
}
```

Suppose that f1 and f2 are two objects of Foo.

Invoking `f1.setI(10)` is to execute `this.i = 10`, where `this` refers f1

Invoking `f2.setI(45)` is to execute `this.i = 45`, where `this` refers f2
public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public Circle() {
        this(1.0);
    }

    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }
}

Every instance variable belongs to an instance represented by this, which is normally omitted.
Class Abstraction and Encapsulation

Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.

Class implementation is like a black box hidden from the clients

Class Contract (Signatures of public methods and public constants)

Clients use the class through the contract of the class
## Designing the Loan Class

<table>
<thead>
<tr>
<th>Loan</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>annualInterestRate: double</td>
<td>The annual interest rate of the loan (default: 2.5).</td>
</tr>
<tr>
<td>numberOfYears: int</td>
<td>The number of years for the loan (default: 1).</td>
</tr>
<tr>
<td>loanAmount: double</td>
<td>The loan amount (default: 1000).</td>
</tr>
<tr>
<td>loanDate: Date</td>
<td>The date this loan was created.</td>
</tr>
</tbody>
</table>

- **Loan()**
  Constructs a default Loan object.

- **Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)**
  Constructs a loan with specified interest rate, years, and loan amount.

- **getAnnualInterestRate(): double**
  Returns the annual interest rate of this loan.

- **getNumberOfYears(): int**
  Returns the number of the years of this loan.

- **getLoanAmount(): double**
  Returns the amount of this loan.

- **getLoanDate(): Date**
  Returns the date of the creation of this loan.

- **setAnnualInterestRate(annualInterestRate: double): void**
  Sets a new annual interest rate to this loan.

- **setNumberOfYears(numberOfYears: int): void**
  Sets a new number of years to this loan.

- **setLoanAmount(loanAmount: double): void**
  Sets a new amount to this loan.

- **getMonthlyPayment(): double**
  Returns the monthly payment of this loan.

- **getTotalPayment(): double**
  Returns the total payment of this loan.
// Loan Class
public class Loan {
    private double annualInterestRate;
    private int numberOfYears;
    private double loanAmount;
    private java.util.Date loanDate;

    /** Default constructor */
    public Loan() {
        this(2.5, 1, 1000);
    }

    /** Construct a loan with specified annual interest rate, number of years and loan amount */
    public Loan(double annualInterestRate, int numberOfYears, double loanAmount) {
        this.annualInterestRate = annualInterestRate;
        this.numberOfYears = numberOfYears;
        this.loanAmount = loanAmount;
        loanDate = new java.util.Date();
    }

    /** Return annualInterestRate */
    public double getAnnualInterestRate() {
        return annualInterestRate;
    }
}
// Loan Class continues ...

/** Set a new annualInterestRate */
public void setAnnualInterestRate(double annualInterestRate) {
    this.annualInterestRate = annualInterestRate;
}

/** Return numberOfYears */
public int getNumberOfYears() {
    return numberOfYears;
}

/** Set a new numberOfYears */
public void setNumberOfYears(int numberOfYears) {
    this.numberOfYears = numberOfYears;
}

/** Return loanAmount */
public double getLoanAmount() {
    return loanAmount;
}

/** Set a new loanAmount */
public void setLoanAmount(double loanAmount) {
    this.loanAmount = loanAmount;
}

/** Find monthly payment */
public double getMonthlyPayment() {
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
    (Math.pow(1 / (1 + monthlyInterestRate),
        numberOfYears * 12)));
    return monthlyPayment;
}

/** Find total payment */
public double getTotalPayment() {
    double totalPayment = getMonthlyPayment() * numberOfYears * 12;
    return totalPayment;
}

/** Return loan date */
public java.util.Date getLoanDate() {
    return loanDate;
}
import java.util.Scanner;
public class TestLoanClass {
    /** Main method */
    public static void main(String[] args) {
        // Create a Scanner
        Scanner input = new Scanner(System.in);
        // Enter yearly interest rate
        System.out.print("Enter yearly interest rate, for example, 8.25: ");
        double annualInterestRate = input.nextDouble();
        // Enter number of years
        System.out.print("Enter number of years as an integer: ");
        int numberOfYears = input.nextInt();
        // Enter loan amount
        System.out.print("Enter loan amount, for example, 120000.95: ");
        double loanAmount = input.nextDouble();
        // Create Loan object
        Loan loan =
                new Loan(annualInterestRate, numberOfYears, loanAmount);
        // Display loan date, monthly payment, and total payment
        System.out.printf("The loan was created on %s\n" +
                "The monthly payment is %.2f\nThe total payment is %.2f\n", loan.getLoanDate().toString(), loan.getMonthlyPayment(), loan.getTotalPayment());
    }
}
Object-Oriented Thinking

Chapters 1-6 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. The studies of these techniques lay a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software. This section improves the solution for a problem introduced in Chapter 3 using the object-oriented approach. From the improvements, you will gain the insight on the differences between the procedural programming and object-oriented programming and see the benefits of developing reusable code using objects and classes.
The BMI Class

<table>
<thead>
<tr>
<th>BMI</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-name: String</td>
<td></td>
</tr>
<tr>
<td>-age: int</td>
<td></td>
</tr>
<tr>
<td>-weight: double</td>
<td></td>
</tr>
<tr>
<td>-height: double</td>
<td></td>
</tr>
</tbody>
</table>

+BMI(name: String, age: int, weight: double, height: double)

+BMI(name: String, weight: double, height: double)

+getBMI(): double

+getStatus(): String

The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The name of the person.
The age of the person.
The weight of the person in pounds.
The height of the person in inches.

Creates a BMI object with the specified name, age, weight, and height.

Creates a BMI object with the specified name, weight, height, and a default age 20.

Returns the BMI

Returns the BMI status (e.g., normal, overweight, etc.)
//The BMI Class  (BMI.java)
public class BMI {
    private String name;
    private int age;
    private double weight; // in pounds
    private double height; // in inches
    public static final double KILOGRAMS_PER_POUND = 0.45359237;
    public static final double METERS_PER_INCH = 0.0254;

    public BMI(String name, int age, double weight, double height) {
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.height = height;
    }
    public BMI(String name, double weight, double height) {
        this(name, 20, weight, height);
    }
    public BMI(String name, int age, double weight, double height) {
        this(name, age, weight, height);
    }
    public BMI(String name, double weight, double height) {
        this(name, 20, weight, height);
    }
    public double getBMI() {
        double bmi = weight * KILOGRAMS_PER_POUND /
                     ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
        return Math.round(bmi * 100) / 100.0;
    }
}
//The BMI Class continues..
public String getStatus() {
    double bmi = getBMI();
    if (bmi < 16)
        return "seriously underweight";
    else if (bmi < 18)
        return "underweight";
    else if (bmi < 24)
        return "normal weight";
    else if (bmi < 29)
        return "over weight";
    else if (bmi < 35)
        return "seriously over weight";
    else
        return "gravely over weight";
}
public String getName() {
    return name;
}
public int getAge() {
    return age;
}
// UseBMIClass.java Main class

public class UseBMIClass {
    public static void main(String[] args) {

        BMI bmi1 = new BMI("John Doe", 18, 145, 70);
        System.out.println("The BMI for " + bmi1.getName() + " is " + bmi1.getBMI() + " " + bmi1.getStatus());

        BMI bmi2 = new BMI("Peter King", 215, 70);
        System.out.println("The BMI for " + bmi2.getName() + " is " + bmi2.getBMI() + " " + bmi2.getStatus());
    }
}
### Example: The Course Class

<table>
<thead>
<tr>
<th>Course</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name: String</td>
<td>The name of the course.</td>
</tr>
<tr>
<td>-students: String[]</td>
<td>The students who take the course.</td>
</tr>
<tr>
<td>-numberOfStudents: int</td>
<td>The number of students (default: 0).</td>
</tr>
<tr>
<td>+Course(name: String)</td>
<td>Creates a Course with the specified name.</td>
</tr>
<tr>
<td>+getName(): String</td>
<td>Returns the course name.</td>
</tr>
<tr>
<td>+addStudent(student: String): void</td>
<td>Adds a new student to the course list.</td>
</tr>
<tr>
<td>+getStudents(): String[]</td>
<td>Returns the students for the course.</td>
</tr>
<tr>
<td>+getNumberOfStudents(): int</td>
<td>Returns the number of students for the course.</td>
</tr>
</tbody>
</table>
// The Course Class  (Course.java)
public class Course {
    private String courseName;
    private String[] students = new String[100];
    private int numberOfStudents;

    public Course(String courseName) {
        this.courseName = courseName;
    }

    public void addStudent(String student) {
        students[numberOfStudents] = student;
        numberOfStudents++;
    }

    public String[] getStudents() {
        return students;
    }

    public int getNumberOfStudents() {
        return numberOfStudents;
    }

    public String getCourseName() {
        return courseName;
    }

    public void dropStudent(String student) {
        // Left as an exercise in Exercise 9.9
    }
}
public class TestCourse {
    public static void main(String[] args) {
        Course course1 = new Course("Data Structures");
        Course course2 = new Course("Database Systems");

        course1.addStudent("Peter Jones");
        course1.addStudent("Brian Smith");
        course1.addStudent("Anne Kennedy");

        course2.addStudent("Peter Jones");
        course2.addStudent("Steve Smith");

        System.out.println("Number of students in course1: ", course1.getNumberofStudents());
        String[] students = course1.getStudents();
        for (int i = 0; i < course1.getNumberofStudents(); i++)
            System.out.print(students[i] + ", ");

        System.out.println();
        System.out.println("Number of students in course2: ", course2.getNumberofStudents());
    }
}
Designing a Class

(Coherence) A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.
(Separating responsibilities) A single entity with too many responsibilities can be broken into several classes to separate responsibilities. The classes `String`, `StringBuilder`, and `StringBuffer` all deal with strings, for example, but have different responsibilities. The `String` class deals with immutable strings, the `StringBuilder` class is for creating mutable strings, and the `StringBuffer` class is similar to `StringBuilder` except that `StringBuffer` contains synchronized methods for updating strings.
Designing a Class, cont.

- Classes are designed for reuse. Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it, design the properties to ensure that the user can set properties in any order, with any combination of values, and design methods to function independently of their order of occurrence.
Designing a Class, cont.

- Provide a public no-arg constructor and override the `equals` method and the `toString` method defined in the `Object` class whenever possible.
Designing a Class, cont.

- Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. Always place the data declaration before the constructor, and place constructors before methods. Always provide a constructor and initialize variables to avoid programming errors.
Using Visibility Modifiers

- Each class can present two contracts – one for the users of the class and one for the extenders of the class. Make the fields private and accessor methods public if they are intended for the users of the class. Make the fields or method protected if they are intended for extenders of the class. The contract for the extenders encompasses the contract for the users. The extended class may increase the visibility of an instance method from protected to public, or change its implementation, but you should never change the implementation in a way that violates that contract.
Using Visibility Modifiers, cont.

- A class should use the private modifier to hide its data from direct access by clients. You can use get methods and set methods to provide users with access to the private data, but only to private data you want the user to see or to modify. A class should also hide methods not intended for client use. The gcd method in the Rational class in Example 11.2, “The Rational Class,” is private, for example, because it is only for internal use within the class.
Using the static Modifier

- A property that is shared by all the instances of the class should be declared as a static property.