Object-Oriented Thinking (Part 1)

Introduction to Programming and Computational Problem Solving - 2
CSE 8B
Lecture 8
Announcements

• Assignment 3 is due today, 11:59 PM
• Quiz 3 is Oct 22
• Assignment 4 will be released today
  – Due Oct 27, 11:59 PM
• Educational research study
  – Oct 22, weekly survey
• Reading
  – Liang
    • Chapter 10
Object-oriented thinking

- The advantages of object-oriented programming over procedural programming
- Classes provide more flexibility and modularity for building reusable software
- How to solve problems using the object-oriented paradigm
- Class design
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 lets the user know how the class can be used
  - The *class contract*
- 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 encapsulation**
  - A class is called an *abstract data type* (ADT)

Class implementation is like a black box hidden from the clients

Class

Class Contract (Signatures of public methods and public constants)

Clients use the class through the contract of the class
Class abstraction and encapsulation

• For example, a class for a loan

<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>
<tr>
<td>+Loan()</td>
<td>Constructs a default Loan object.</td>
</tr>
<tr>
<td>+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)</td>
<td>Constructs a loan with specified interest rate, years, and loan amount.</td>
</tr>
<tr>
<td>+getAnnualInterestRate(): double</td>
<td>Returns the annual interest rate of this loan.</td>
</tr>
<tr>
<td>+getNumberOfYears(): int</td>
<td>Returns the number of the years of this loan.</td>
</tr>
<tr>
<td>+getLoanAmount(): double</td>
<td>Returns the amount of this loan.</td>
</tr>
<tr>
<td>+getLoanDate(): Date</td>
<td>Returns the date of the creation of this loan.</td>
</tr>
<tr>
<td>+setAnnualInterestRate(annualInterestRate: double): void</td>
<td>Sets a new annual interest rate to this loan.</td>
</tr>
<tr>
<td>+setNumberOfYears(numberOfYears: int): void</td>
<td>Sets a new number of years to this loan.</td>
</tr>
<tr>
<td>+setLoanAmount(loanAmount: double): void</td>
<td>Sets a new amount to this loan.</td>
</tr>
<tr>
<td>+getMonthlyPayment(): double</td>
<td>Returns the monthly payment of this loan.</td>
</tr>
<tr>
<td>+getTotalPayment(): double</td>
<td>Returns the total payment of this loan.</td>
</tr>
</tbody>
</table>

The creator of the class provides a description of the class and lets the user know how the class can be used.

The class contract
Class abstraction and encapsulation

• A class is designed for use by many different users (or customers or clients)
• To be useful in a wide range of applications, a class should provide a variety of ways for customization through properties, and constructors and methods that, together, are minimal and complete
Thinking in objects

• Procedural programming focuses on designing methods

• Object-oriented programming
  – Couples data and methods together into objects
  – Focuses on designing objects and operations on objects

• Object-orientated programming combines the power of procedural programming with an additional component that integrates data with operations into objects
Procedural programming vs object-oriented programming

• Procedural programming
  – Data and operations on data are separate
  – Requires passing data to methods

• Object-oriented programming
  – Data and operations on data are in an object
  – Organizes programs like the real world
    • All objects are associated with both attributes and activities
  – Using objects improves software reusability and makes programs easier to both develop and maintain
Class relationships

• To design classes, one must understand the relationships among classes
  – Association
  – Aggregation
  – Composition
  – Inheritance (covered next week)
Association

• A general binary relationship that describes an activity between two classes

• For example
  – A student taking course is an association between the Student class and the Course class
  – A faculty member teaching a course is an association between the Faculty class and the Course class
Association

• Multiplicity
  – The number of objects of a class

• For example
  – Each student may take any number (*) of courses
  – Each course must have 5 to 60 students
  – Each course is taught by 1 faculty member
  – Each faculty member must teach 0 to 3 courses
Association

• In Java, associations can be implemented using data fields and methods
  – For example
    • A student takes a course
      addCourse method in Student class
      addStudent method in Course class
    • A faculty member teaches a course
      addCourse method in Faculty class
      setFaculty method in Course class
    • The Student class may store the courses a student is taking
      private Course[] courseList;
    • The Faculty class may store the courses a faculty member is teaching
      private Course[] courseList;
  • There are many possible ways to implement association relationships
Aggregation

• Special form of association representing an owner-subject relationship
  – The owner object is called an aggregating object and its class is called an aggregating class
  – The subject object is called an aggregated object and its class is called an aggregated class

• Models has-a relationships
  – For example
    • A student has-a name
    • A student has-an address
Composition

• Aggregation between two objects is called *composition* if the existence of the aggregated object is dependent on the aggregating object
  – Exclusive ownership of the subject
  – The subject (i.e., aggregated object) cannot (conceptually) exist on its own
• For example
  – A book *has-a* page and when the book is destroyed, so is the page
  – A page has no meaning or purpose without the book
Aggregation and composition

• For example
  – When the student object is destroyed
    • Their name is destroyed (composition)
    • Their address is not destroyed (aggregation)

Each address is shared by up to 3 students
Aggregation and composition

- Usually represented as a data field in the aggregating class

```java
public class Name {
    ...
}

public class Student {
    private Name name;
    private Address address;
    ...
}

public class Address {
    ...
}
```

Aggregated class
Aggregating class
Aggregated class
Aggregation between same class

- Aggregation may exist between objects of the same class
  - For example, a person may have a supervisor
    ```java
    public class Person {
        // The type for the data is the class itself
        private Person supervisor;
        ...
    }
    ```
  - For example, a person may have multiple supervisors
    ```java
    public class Person {
        // The type for the data is the class itself
        private Person[] supervisors;
        ...
    }
    ```
Aggregation or composition

• Warning: Since aggregation and composition relationships are represented using classes in similar ways, many texts do not differentiate them, calling both compositions
For example, a class for a course

<table>
<thead>
<tr>
<th>Course</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-courseName: String</td>
<td>The name of the course.</td>
</tr>
<tr>
<td>-students: String[]</td>
<td>An array to store the students for the course.</td>
</tr>
<tr>
<td>-numberOfStudents: int</td>
<td>The number of students (default: 0).</td>
</tr>
<tr>
<td>+Course(courseName: String)</td>
<td>Creates a course with the specified name.</td>
</tr>
<tr>
<td>+getCourseName(): String</td>
<td>Returns the course name.</td>
</tr>
<tr>
<td>+addStudent(student: String): void</td>
<td>Adds a new student to the course.</td>
</tr>
<tr>
<td>+dropStudent(student: String): void</td>
<td>Drops a student from the course.</td>
</tr>
<tr>
<td>+getStudents(): String[]</td>
<td>Returns the students in the course.</td>
</tr>
<tr>
<td>+getNumberOfStudents(): int</td>
<td>Returns the number of students in the course.</td>
</tr>
</tbody>
</table>
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.print("Number of students in course2: 
        + course2.getNumberOfStudents());
    }
}
public class Course {
    private String courseName;
    private String[] students = new String[4];
    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 10.9
    }
}
Class design and development

• Use a UML class diagram to design the class
• Write a test program that uses the class
  – Developing a class and using a class are two separate tasks
  – It is easier to implement a class if you must use the class
• Implement the class
• Use Javadoc to document the class (contract)
Object-oriented thinking

• Classes provide more flexibility and modularity for building reusable software

• Class abstraction and encapsulation
  – 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
Next Lecture

• Object-oriented thinking
• Reading
  – Liang
    • Chapter 10