Object-Oriented Thinking

Introduction to Programming and Computational Problem Solving - 2
CSE 8B
Lecture 9
Announcements

• Assignment 4 is due Oct 26, 11:59 PM
• Quiz 4 is Oct 28
• Assignment 5 will be released Oct 26
  – Due Nov 2, 11:59 PM
• Educational research study
  – Oct 28, 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
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
Procedural programming: method abstraction

- You can think of the method body as a black box that contains the detailed implementation for the method.
Object-oriented programming: 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 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(</td>
<td>Sets a new annual interest rate to this loan.</td>
</tr>
<tr>
<td>annualInterestRate: double): void</td>
<td></td>
</tr>
<tr>
<td>+setNumberOfYears(</td>
<td>Sets a new number of years to this loan.</td>
</tr>
<tr>
<td>numberOfYears: int): void</td>
<td></td>
</tr>
<tr>
<td>+setLoanAmount(</td>
<td>Sets a new amount to this loan.</td>
</tr>
<tr>
<td>loanAmount: double): void</td>
<td></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
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
Class design and development

- For example, a class for a course

<table>
<thead>
<tr>
<th>Course</th>
<th></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.println("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) {
        // TODO
    }
}

<table>
<thead>
<tr>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>-courseName: String</td>
</tr>
<tr>
<td>-students: String[]</td>
</tr>
<tr>
<td>-numberOfStudents: int</td>
</tr>
<tr>
<td>+Course(courseName: String)</td>
</tr>
<tr>
<td>+getCourseName(): String</td>
</tr>
<tr>
<td>+addStudent(student: String): void</td>
</tr>
<tr>
<td>+dropStudent(student: String): void</td>
</tr>
<tr>
<td>+getStudents(): String[]</td>
</tr>
<tr>
<td>+getNumberOfStudents(): int</td>
</tr>
</tbody>
</table>
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
Primitive data type values as objects

• A primitive data type is not an object
• But it can be wrapped in an object using a Java API wrapper class
  
  Boolean
  Character (not Char)
  Short
  Byte
  Integer (not Int)
  Long
  Float
  Double

Notes
• The wrapper classes do not have no-arg constructors
• The instances of all wrapper classes are immutable (i.e., their internal values cannot be changed once the objects are created)
# Integer and Double wrapper classes

<table>
<thead>
<tr>
<th>java.lang.Integer</th>
<th>java.lang.Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: int</td>
<td>value: double</td>
</tr>
<tr>
<td>+MAX_VALUE: int</td>
<td>+MAX_VALUE: double</td>
</tr>
<tr>
<td>+MIN_VALUE: int</td>
<td>+MIN_VALUE: double</td>
</tr>
<tr>
<td>+Integer(value: int)</td>
<td>+Double(value: double)</td>
</tr>
<tr>
<td>+Integer(s: String)</td>
<td>+Double(s: String)</td>
</tr>
<tr>
<td>+byteValue(): byte</td>
<td>+byteValue(): byte</td>
</tr>
<tr>
<td>+shortValue(): short</td>
<td>+shortValue(): short</td>
</tr>
<tr>
<td>+intValue(): int</td>
<td>+intValue(): int</td>
</tr>
<tr>
<td>+longValue(): long</td>
<td>+longValue(): long</td>
</tr>
<tr>
<td>+floatValue(): float</td>
<td>+floatValue(): float</td>
</tr>
<tr>
<td>+doubleValue(): double</td>
<td>+doubleValue(): double</td>
</tr>
<tr>
<td>+compareTo(o: Integer): int</td>
<td>+compareTo(o: Double): int</td>
</tr>
<tr>
<td>+toString(): String</td>
<td>+toString(): String</td>
</tr>
<tr>
<td>+valueOf(s: String): Integer</td>
<td>+valueOf(s: String): Double</td>
</tr>
<tr>
<td>+valueOf(s: String, radix: int): Integer</td>
<td>+valueOf(s: String, radix: int): Double</td>
</tr>
<tr>
<td>+parseInt(s: String): int</td>
<td>+parseInt(s: String): int</td>
</tr>
<tr>
<td>+parseInt(s: String, radix: int): int</td>
<td>+parseInt(s: String, radix: int): int</td>
</tr>
</tbody>
</table>

CSE 8B, Fall 2022
Wrapper classes

• Constructors
• Class Constants MAX_VALUE and MIN_VALUE
• Conversion Methods
Numeric wrapper class constructors

• You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value
  – For example, the constructors for Integer and Double are
    public Integer(int value)
    public Integer(String s)
    public Double(double value)
    public Double(String s)
Numeric wrapper class constants

• Each numerical wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`

• `MAX_VALUE` represents the maximum value of the corresponding primitive data type

• For Byte, Short, Integer, and Long, `MIN_VALUE` represents the minimum byte, short, int, and long values

• For Float and Double, `MIN_VALUE` represents the minimum **positive** float and double values
Numeric wrapper class conversion methods

• Each numeric wrapper class implements the abstract methods `doubleValue`, `floatValue`, `intValue`, `longValue`, and `shortValue`
  – Defined in the abstract `Number` class (covered in three weeks)

• These methods “convert” objects into primitive type values
Numeric wrapper class
static valueOf methods

• The numeric wrapper classes have a useful class method valueOf(String s)
• This method creates a new object initialized to the value represented by the specified string
  – For example
    
    ```java
    Double doubleObject = Double.valueOf("12.4");
    Integer integerObject = Integer.valueOf("12");
    ```
Numeric wrapper class static parsing methods

• Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on 10 or any specified radix (e.g., 2 for binary, 8 for octal, 10 for decimal, 16 for hexadecimal)
  – For example
    
    `Integer.parseInt("13")` returns 13
    `Integer.parseInt("13", 10)` returns 13
    `Integer.parseInt("1A", 16)` returns 26
Automatic conversion between primitive types and wrapper class types

• Converting a primitive value to a wrapper object is called *boxing*

• Converting a wrapper object to a primitive value is called *unboxing*

• The Java compiler will automatically convert a primitive data type value to an object using a wrapper class (*autoboxing*) and vice versa (*autounboxing*), depending on the context
Automatic conversion between primitive types and wrapper class types

Integer[] intArray = {new Integer(2),
    new Integer(4), new Integer(3)};

Equivalent

Integer[] intArray = {2, 4, 3};

New JDK 1.5 boxing

Integer[] intArray = {new Integer(2),
    new Integer(4), new Integer(3)};

(a)

(b)

Autoboxing

Integer[] intArray = {1, 2, 3};

System.out.println(intArray[0] + intArray[1] + intArray[2]);

Autounboxing
BigInteger and BigDecimal classes

• If you need to compute with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes in the java.math package
• Both are immutable
• Both extend the Number class and implement the Comparable interface (covered in three weeks)
BigInteger and BigDecimal classes

BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);

BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);

Scale
String class

• The String class has 13 constructors and more than 40 methods
• A good example for learning classes and objects
Constructing strings

• Create from a string literal
  – Syntax
    ```java
    String newString = new String(stringLiteral);
    ```
  – Example
    ```java
    String message = new String("Welcome to Java");
    ```
  – Since strings are used frequently, Java provides a shorthand initializer for creating a string
    ```java
    String message = "Welcome to Java";
    ```

• Create from an array of characters
  – Syntax
    ```java
    String newString = new String(charArray);
    ```
  • where, for example
    ```java
    char[] charArray = {'C', 'S', 'E', ' ', '8', 'B'};
    ```
Strings are immutable

• A String object is immutable (i.e., its contents cannot be changed once the string is created)

• The following code does not change the contents of the string

  String s = "Java";
  s = "HTML";
Strings are immutable

```java
String s = "Java";
s = "HTML";
```

After executing `String s = "Java";`

```
String s
: String
String object for "Java"
```

Contents cannot be changed

After executing `s = "HTML";`

```
String s
: String
String object for "Java"
```
```
: String
String object for "HTML"
```

This string object is now unreferenced
Strings are immutable

```java
String s = "Java";
s = "HTML";
```

After executing `String s = "Java";`

```
<table>
<thead>
<tr>
<th>s</th>
<th>String</th>
</tr>
</thead>
</table>
|   | String object for "Java"
```

Contents cannot be changed

After executing `s = "HTML";`

```
<table>
<thead>
<tr>
<th>s</th>
<th>String</th>
</tr>
</thead>
</table>
|   | String object for "Java"
|   | String object for "HTML"
```

This string object is now unreferenced
Interned strings

• Since strings are immutable and are frequently used, to improve efficiency and save memory, the Java Virtual Machine (JVM) uses a unique instance for string literals with the same character sequence

• Such an instance is called *interned*
Interned strings

• A new object is created if you use the new operator

• If you use the string initializer, no new object is created if the interned object is already created

```java
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```

s1 == s2 is false
s1 == s3 is true
Replacing and splitting strings

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace(char oldChar, char newChar)</td>
<td>Returns a new string that replaces all matching character in this string with the new character.</td>
</tr>
<tr>
<td>replaceFirst(String oldString, String newString)</td>
<td>Returns a new string that replaces the first matching substring in this string with the new substring.</td>
</tr>
<tr>
<td>replaceAll(String oldString, String newString)</td>
<td>Returns a new string that replace all matching substrings in this string with the new substring.</td>
</tr>
<tr>
<td>split(String delimiter)</td>
<td>Returns an array of strings consisting of the substrings split by the delimiter.</td>
</tr>
</tbody>
</table>
Replacing a string

- "Welcome".replace('e', 'A') returns a new string WAlcomA
- "Welcome".replaceFirst("e", "AB") returns a new string WAB1come
- "Welcome".replace("e", "AB") returns a new string WAB1comAB
- "Welcome".replace("el", "AB") returns a new string WABcome
Splitting a string

• Split a string into an array of strings
  – For example, using # as a delimiter
    ```java
    String[] tokens = "CSE#8B#uses#Java".split("#", 0);
    for (int i = 0; i < tokens.length; i++)
      System.out.print(tokens[i] + " ");
    ```
  – Displays CSE 8B uses Java
Matching, replacing, and splitting by patterns

• You can match, replace, or split a string by specifying a pattern
  – For example
    "Java".equals("Java");
    "Java".matches("Java");

• This is an extremely useful and powerful feature known as regular expression
  – Liang, appendix H
  – https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html#sum
  – https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html#sum
Convert character and numbers to strings

• The String class provides several static `valueOf` methods for converting a character, an array of characters, and numeric values to strings

• These methods have the same name `valueOf` with different argument types `char`, `char[]`, `double`, `long`, `int`, and `float`
  
  – For example, to convert a `double` value to a string, use `String.valueOf(5.44)`
    
    • The return value is string consists of characters '5', '.', '4', and '4'
    
    • Compare with `String s = 5.44 + "";`
StringBuilder and StringBuffer classes

• The StringBuilder and StringBuffer classes are alternatives to the String class
• In general, a StringBuilder or StringBuffer can be used wherever a string is used
• StringBuilder and StringBuffer are more flexible than String
• You can add, insert, or append new contents into a string buffer, whereas the value of a String object is fixed once the string is created
## StringBuilder constructors

<table>
<thead>
<tr>
<th>java.lang.StringBuilder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+StringBuilder()</td>
<td>Constructs an empty string builder with capacity 16.</td>
</tr>
<tr>
<td>+StringBuilder(capacity: int)</td>
<td>Constructs a string builder with the specified capacity.</td>
</tr>
<tr>
<td>+StringBuilder(s: String)</td>
<td>Constructs a string builder with the specified string.</td>
</tr>
</tbody>
</table>
## Modifying strings in the builder

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+append(data: char[]): StringBuilder</td>
<td>Appends a char array into this string builder.</td>
</tr>
<tr>
<td>+append(data: char[], offset: int, len: int): StringBuilder</td>
<td>Appends a subarray in data into this string builder.</td>
</tr>
<tr>
<td>+append(v: aPrimitiveType): StringBuilder</td>
<td>Appends a primitive type value as a string to this builder.</td>
</tr>
<tr>
<td>+append(s: String): StringBuilder</td>
<td>Appends a string to this string builder.</td>
</tr>
<tr>
<td>+delete(startIndex: int, endIndex: int): StringBuilder</td>
<td>Deletes characters from startIndex to endIndex.</td>
</tr>
<tr>
<td>+deleteCharAt(index: int): StringBuilder</td>
<td>Deletes a character at the specified index.</td>
</tr>
<tr>
<td>+insert(index: int, data: char[], offset: int, len: int): StringBuilder</td>
<td>Inserts a subarray of the data in the array to the builder at the specified index.</td>
</tr>
<tr>
<td>+insert(offset: int, data: char[]): StringBuilder</td>
<td>Inserts data into this builder at the position offset.</td>
</tr>
<tr>
<td>+insert(offset: int, b: aPrimitiveType): StringBuilder</td>
<td>Inserts a value converted to a string into this builder.</td>
</tr>
<tr>
<td>+insert(offset: int, s: String): StringBuilder</td>
<td>Inserts a string into this builder at the position offset.</td>
</tr>
<tr>
<td>+replace(startIndex: int, endIndex: int, s: String): StringBuilder</td>
<td>Replaces the characters in this builder from startIndex to endIndex with the specified string.</td>
</tr>
<tr>
<td>+reverse(): StringBuilder</td>
<td>Reverses the characters in the builder.</td>
</tr>
<tr>
<td>+setCharAt(index: int, ch: char): void</td>
<td>Sets a new character at the specified index in this builder.</td>
</tr>
</tbody>
</table>
### The toString, capacity, length, setLength, and charAt methods

<table>
<thead>
<tr>
<th>java.lang.StringBuilder</th>
</tr>
</thead>
<tbody>
<tr>
<td>+toString(): String</td>
</tr>
<tr>
<td>+capacity(): int</td>
</tr>
<tr>
<td>+charAt(index: int): char</td>
</tr>
<tr>
<td>+length(): int</td>
</tr>
<tr>
<td>+setLength(newLength: int): void</td>
</tr>
<tr>
<td>+substring(startIndex: int): String</td>
</tr>
<tr>
<td>+substring(startIndex: int, endIndex: int): String</td>
</tr>
<tr>
<td>+trimToSize(): void</td>
</tr>
</tbody>
</table>

- `toString()`: Returns a string object from the string builder.
- `capacity()`: Returns the capacity of this string builder.
- `charAt(index: int)`: Returns the character at the specified index.
- `length()`: Returns the number of characters in this builder.
- `setLength(newLength: int)`: Sets a new length in this builder.
- `substring(startIndex: int)`: Returns a substring starting at startIndex.
- `substring(startIndex: int, endIndex: int)`: Returns a substring from startIndex to endIndex - 1.
- `trimToSize()`: Reduces the storage size used for the string builder.
Next Lecture

• Inheritance

• Reading
  – Liang
    • Chapter 11