Thinking in Objects

CSE 114, Computer Science 1
SUNY Korea

Pradipta De
pradipta.de@sunykorea.ac.kr

Acknowledgement for the slides: Dr. Paul Fodor (CS Stony Brook)
Immutable Objects and Classes

- Immutable object: the contents of an object cannot be changed once the object is created - its class is called an immutable class
- Example immutable class: no set method in the Circle class
  ```java
  public class Circle{
      private double radius;
      public Circle() { }
      public Circle(double radius) { this.radius = radius; }
      public double getRadius() { return radius; }
  }
  ```
  - 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!
Example mutable

```java
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 class Test {
    public static void main(String[] args) {
        Student student = new Student(111223333, 1998, 5, 3);
        BirthDate date = student.getBirthDate();
        date.setYear(2014); // Now the student birth year is changed!
    }
}
```
What Class is Immutable?

1. It must mark all data fields private!
2. Provide no mutator methods!
3. Provide no accessor methods that would return a reference to a mutable data field object!
Scope of Variables

- Data Field Variables can be declared anywhere inside a class
  - The scope of instance and static variables is the entire class!
  - Initialized with default values.
- 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.
Example of scoping in class

```java
public class F {
    private int x = 0; // Instance variable
    private int y = 0;

    public F() {
    }

    public void p() {
        int x = 1; // Local variable
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }
}
```

What is the output of f.p()?
The **this** Keyword

- The **this** keyword is the name of a reference that refers to an object itself.

- Common uses of the **this** keyword:
  1. Reference a class’s “hidden” data fields
  2. To enable a constructor to invoke another constructor of the same class as the first statement in the constructor.
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 to `f1`.

Invoking `f2.setI(45)` is to execute `this.i = 45`, where `this` refers to `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.
What is wrong with this code?

```java
public class CC {
    private int p;

    public CC() {
        this(0);
    }

    public CC(int p) {
        p = p;
    }

    public void setP(int p) {
        p = p;
    }
}
```
Abstraction = 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 does not need to know how the class is implemented: it is encapsulated and hidden.
## Designing the Loan Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>

### Attributes
- `annualInterestRate: double` - The annual interest rate of the loan (default: 2.5).
- `numberOfYears: int` - The number of years for the loan (default: 1).
- `loanAmount: double` - The loan amount (default: 1000).
- `loanDate: Date` - The date this loan was created.
public class Loan {
    private double annualInterestRate;
    private int numberOfYears;
    private double loanAmount;
    private java.util.Date loanDate;
    public Loan() {
        this(2.5, 1, 1000);
    }
    public Loan(double annualInterestRate, int numberOfYears,
            double loanAmount) {
        this.annualInterestRate = annualInterestRate;
        this.numberOfYears = numberOfYears;
        this.loanAmount = loanAmount;
        loanDate = new java.util.Date();
    }
    public double getMonthlyPayment() {
        double monthlyInterestRate = annualInterestRate / 1200;
        double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
                     (Math.pow(1 / (1 + monthlyInterestRate), numberOfYears * 12)));
        return monthlyPayment;
    }
    public double getTotalPayment() {
        double totalPayment = getMonthlyPayment() * numberOfYears * 12;
        return totalPayment;
    }
} ...
Compute BMI: Procedural Program

```java
import java.util.Scanner;

public class ComputeAndInterpretBMI {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter weight in pounds
        System.out.print("Enter weight in pounds: ");
        double weight = input.nextDouble();

        // Prompt the user to enter height in inches
        System.out.print("Enter height in inches: ");
        double height = input.nextDouble();

        final double KILOGRAMS_PER_POUND = 0.45359237; // Constant
        final double METERS_PER_INCH = 0.0254; // Constant

        // Compute BMI
        double weightInKilograms = weight * KILOGRAMS_PER_POUND;
        double heightInMeters = height * METERS_PER_INCH;
        double bmi = weightInKilograms / heightInMeters;

        // Display result
        System.out.println("BMI is " + bmi);
        if (bmi < 18.5)
            System.out.println("Underweight");
        else if (bmi < 25)
            System.out.println("Normal");
        else if (bmi < 30)
            System.out.println("Overweight");
        else
            System.out.println("Obese");
    }
}
```

To make the code reusable, we can define a method that can be invoked with different parameter values:

```java
public static double getBMI(double weight, double height)
```

How do I associate the BMI with a person’s name and birthday?
The BMI Class

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.)
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 double getBMI() {
        double bmi = weight * KILOGRAMS_PER_POUND / ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
        return Math.round(bmi * 100) / 100.0;
    }

    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;  }
    public double getWeight() {    return weight;  }
    public double getHeight() {    return height;  }
}
• Procedural Paradigm focuses on designing methods

• Object oriented paradigm couples data and methods together into objects
Class Relationships

- Association
  - Shows activity between two classes

- Aggregation
  - A special form of association showing ownership relationship between two objects

- Composition
  - If an object is exclusively owned by an aggregating object, then the relationship is called composition

- Inheritance
  - When one object is more general than the other, and has all the properties of the parent object, then it is an inheritance relationship
Association example

```
public class Student {
    private Course[] courseList;

    public void addCourse(Course s) { ... }
}
```

```
public class Course {
    private Student[] classList;
    private Faculty faculty;

    public void addStudent(Student s) { ... }

    public void setFaculty(Faculty faculty) { ... }
}
```

```
public class Faculty {
    private Course[] courseList;

    public void addCourse(Course c) { ... }
}
```
Aggregation and Composition example

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

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

```java
public class Address {
    ...
}
```

Aggregated class

Aggregating class

Aggregated class
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>
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;
    }
}
## Example: The StackOfIntegers Class

<table>
<thead>
<tr>
<th>StackOfIntegers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-elements: int[]</td>
<td>An array to store integers in the stack.</td>
</tr>
<tr>
<td>-size: int</td>
<td>The number of integers in the stack.</td>
</tr>
<tr>
<td>+StackOfIntegers()</td>
<td>Constructs an empty stack with a default capacity of 16.</td>
</tr>
<tr>
<td>+StackOfIntegers(capacity: int)</td>
<td>Constructs an empty stack with a specified capacity.</td>
</tr>
<tr>
<td>+empty(): boolean</td>
<td>Returns true if the stack is empty.</td>
</tr>
<tr>
<td>+peek(): int</td>
<td>Returns the integer at the top of the stack without removing it from the stack.</td>
</tr>
<tr>
<td>+push(value: int): int</td>
<td>Stores an integer into the top of the stack.</td>
</tr>
<tr>
<td>+pop(): int</td>
<td>Removes the integer at the top of the stack and returns it.</td>
</tr>
<tr>
<td>+getSize(): int</td>
<td>Returns the number of elements in the stack.</td>
</tr>
</tbody>
</table>
Designing the **StackOfIntegers** Class

Data1 → Data2 → Data3

Data3 → Data2 → Data1

Data1 → Data2 → Data1

Data3 → Data2 → Data1
Implementing **StackOfIntegers** Class

- `elements[0]`
- `elements[1]`
- `elements[size-1]`
- `capacity`
- `top`
- `bottom`
- `size`
public class StackOfIntegers {
    private int[] elements;
    private int size;
    public static final int DEFAULT_CAPACITY = 16;
    public StackOfIntegers() {
        this(DEFAULT_CAPACITY);
    }
    public StackOfIntegers(int capacity) {
        elements = new int[capacity];
    }
    public void push(int value) {
        if (size >= elements.length) {
            int[] temp = new int[elements.length * 2];
            System.arraycopy(elements, 0, temp, 0, elements.length);
            elements = temp;
        }
        elements[size++] = value;
    }
    public int pop() {
        return elements[--size];
    }
    public int peek() {
        return elements[size - 1];
    }
    public int getSize() {
        return size;
    }
}
Constructing Strings

- Pattern:
  ```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";
  ```
Strings Are Immutable

- A String object is immutable; its contents cannot be changed.

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

After executing `String s = "Java";`

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

  Contents cannot be changed

After executing `s = "HTML";`

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

  This string object is now unreferenced

- String object for "HTML"
Interned Strings

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));

- 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.

display

s1 == s2 is false
s1 == s3 is true
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
  - `valueOf` with different argument types `char`, `char[]`, `double`, `long`, `int`, and `float`
  - Example: convert a double value to a string: `String.valueOf(5.44) → “5.44”`
The **Character Class**

<table>
<thead>
<tr>
<th>java.lang.Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Character(value: char)</td>
</tr>
<tr>
<td>+charValue(): char</td>
</tr>
<tr>
<td>+compareTo(anotherCharacter: Character): int</td>
</tr>
<tr>
<td>+equals(anotherCharacter: Character): boolean</td>
</tr>
<tr>
<td>+isDigit(ch: char): boolean</td>
</tr>
<tr>
<td>+isLetter(ch: char): boolean</td>
</tr>
<tr>
<td>+isLetterOrDigit(ch: char): boolean</td>
</tr>
<tr>
<td>+isLowerCase(ch: char): boolean</td>
</tr>
<tr>
<td>+isUpperCase(ch: char): boolean</td>
</tr>
<tr>
<td>+toLowerCase(ch: char): char</td>
</tr>
<tr>
<td>+toUpperCase(ch: char): char</td>
</tr>
</tbody>
</table>

Constructs a character object with char value

Returns the char value from this object

Compares this character with another

Returns true if this character equals to another

Returns true if the specified character is a digit

Returns true if the specified character is a letter

Returns true if the character is a letter or a digit

Returns true if the character is a lowercase letter

Returns true if the character is an uppercase letter

Returns the lowercase of the specified character

Returns the uppercase of the specified character
Examples

Character charObject = new Character('b');

charObject.compareTo(new Character('a'))
  returns 1
charObject.compareTo(new Character('b'))
  returns 0
charObject.compareTo(new Character('c'))
  returns -1
charObject.compareTo(new Character('d'))
  returns -2
charObject.equals(new Character('b'))
  returns true
charObject.equals(new Character('d'))
  returns false
StringBuilder and StringBuffer

- The **StringBuilder/StringBuffer** class is an alternative to the **String** class:
  - **StringBuilder/StringBuffer** can be used wherever a string is used
  - **StringBuilder/StringBuffer** is 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>
</tr>
</thead>
<tbody>
<tr>
<td>+StringBuilder()</td>
</tr>
<tr>
<td>+StringBuilder(capacity: int)</td>
</tr>
<tr>
<td>+StringBuilder(s: String)</td>
</tr>
</tbody>
</table>

Constructs an empty string builder with capacity 16.
Constructs a string builder with the specified capacity.
Constructs a string builder with the specified string.
# Modifying Strings in the Builder

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

StringBuilder stringBuilder =
    new StringBuilder();
stringBuilder.append("Java");
stringBuilder.insert(2,"HTML and ");
stringBuilder.delete(3, 4);
stringBuilder.deleteCharAt(5);
stringBuilder.reverse();
stringBuilder.replace(4, 8, "HTML");
stringBuilder.setCharAt(0, 'w');
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>
Designing a Class

• Coherence: A class should describe a single entity

• Separating responsibilities: A single entity with too many responsibilities can be broken into several classes to separate responsibilities

• Classes are designed for reuse!

• 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

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