Class Design III:
Good Practices and Bad Practices

Readings:
None

Purpose
➢ Explain how to use inheritance and interfaces appropriately (in Java)

Learning Objectives
➢ Describe the open-closed principle, why it matters, and how it applies to object-oriented code.
➢ Use overloading correctly and recognize inappropriate uses
➢ Explain the Liskov Substitution Principle and recognize when inheritance is not is-a
➢ Describe delegation, recognize its uses and explain how it can be exploited to implement multiple inheritance in Java

Additional References
➢ “Object-Oriented Software Development Using Java”, Xiaoping Jia, Addison Wesley, 2002
➢ “Core Java 2”, Cay Horstmann, Gary Cornell, Sun Microsystems Press, 2003

To Overload or Not to Overload

➢ Overloading: Same name is used for more than one method
➢ Mainly used for convenience
➢ Misuse may reduce program readability
➢ Should use overloading only in two situations:
  ❚
  ❚

Open-Closed Principle

➢ Classes should be open for extension but closed for modification
  ➢ Want to extend the behaviour of our system by adding subclasses
  ➢ without having to modify the superclasses
➢ The principle suggests you should consider possible future subclasses when defining a class
➢ Example:
  ➢

Overloading Examples

Good:

```java
public class StringBuffer
{
    public StringBuffer append(char c)
    {
        //...
    }

    public StringBuffer append(int i)
    {
        //...
    }

    public StringBuffer append(float f)
    {
        //...
    }
}
```

Bad:

```java
public class Employee
{
    public void name(String s)
    {
        //...
    }

    public String name()
    {
        //...
    }
}
```
Is-A Style Inheritance: The right way

- Must be able to replace any instance of a superclass with an instance of any of its subclasses (Liskov Substitution Principle or LSP)
- Each class defines a type (the set of all instances of that class). A subclass following LSP defines a subtype (a subset of the superclass type).
- Example:
  - A Person class and a Student class

Weakening the precondition

- A subclass method can weaken the precondition: when overriding a method from its superclass, a subclass can accept a wider range of values as input.

```java
abstract class Payment {
    /**
     * @pre amt >= 0
     */
    public void setPaymentAmount(int amt) {...}
}

class CashPayment extends Payment { ... }

class CreditCardPayment extends Payment {
    /**
     * @pre true
     */
    public void setPaymentAmount(int amt) {...}
}
```

Strengthening the postcondition

- A subclass method can strengthen the postcondition: a subclass’ method can return a subset of the values returned by the method it overrides.

```java
class Pump {
    /**
     * @post true
     */
    public double volumePumped() {...}
}

class PropanePump extends Pump {
    /**
     * @post value returned is integral and divisible by 5
     */
    public double volumePumped() {...}
}
Is-A Style Inheritance: The right way (cont’d)

- Polymorphic Assignment:
  
  \[ \text{reference}\_\text{to}\_\text{object} = \text{expression}; \]
  
  the type of \text{expression} must be a \text{subtype} of the type of the \text{reference}\_\text{to}\_\text{object}

- You can explicitly convert (cast) references between a type and its subtypes:
  - \text{widening}: converting a subtype to one of its supertypes
  - \text{narrowing or downcasting}: converting a supertype to one of its subtypes

Example: Bank Accounts

Suppose we declare:

```java
public class Account { … }
public class SavingsAccount extends Account { … }
public class CheckingAccount extends Account { … }
```

Then

```java
Account accl, acc2;
accl = new SavingsAccount();
acc2 = new CheckingAccount();
```

Example: Bank Accounts (cont’d)

```java
SavingsAccount acc3;
accl = acc3;
acc3 = accl;
acc3 = (SavingsAccount) accl;
acc3 = (SavingsAccount) acc2;
```

- Similar with arrays:

```java
Account[] acc_array;
SavingsAccount[] s_array = new SavingsAccount[50];

acc_array = s_array;
Account acc4 = acc_array[0];
SavingsAccount saccl = s_array[0];
```

Example: Bank Accounts (cont’d)

```java
BUT
SavingsAccount sac2= acc_array[0];
SavingsAccount sac2= (SavingsAccount)acc_array[0];
```

- Can use \text{instanceof} to check the type of the object:

```java
if (acc2 instanceof SavingsAccount) {
    acc3 = (SavingsAccount) acc2;
} else {
    . . .
```

NOTE: This is generally an indication of bad design. Use polymorphism instead.
More Downcast Uses

- When a method is defined in a subclass only, but is applied to a superclass reference
- Suppose only SavingsAccount defines addInterest()
  then
  Account acc = new SavingsAccount();
  acc.addInterest();

  ((SavingsAccount) acc).addInterest();
or
  SavingsAccount sacc = (SavingsAccount) acc;
  sacc.addInterest();
- When would this situation occur?

Example: Rectangle Class

```java
public class Rectangle {
    private double length;
    private double width;

    public Rectangle(){
        length = 0; width = 0;
    }
    public Rectangle( double l, double w){
        length = l; width = w;
    }

    public void setLength(double l) {
        length = l;
    }

    public void setWidth( double w ){
        width = w;
    }

    public double area() {
        return length * width;
    }
}
```

Limitation Inheritance : The wrong way

- Subclass restricts the behavior inherited from the superclass
- Violates is-a relationship
- Violates Liskov substitution principle
- Usually used for implementation convenience (obviously in the wrong way)
- Examples
  - Square defined as a subclass of Rectangle (next slide)
    - Methods setLength and setWidth are not applicable to a square
  - Circle defined as a subclass of Ellipse
  - etc.
Example: Square Class (the wrong way)

```java
public class Square extends Rectangle {

    public Square() {
        super();
    }

    public Square(double s) {
        super(s, s);
    }
}
```

Delegation

- A method delegates the execution of a task to another object of a different type
- Think of the “other object” as a servant used to carry out the task
- In OO languages delegation can be:
  - class-based (or static)
  - method-based (or dynamic)

Example next slide:
  - `Square` defined using class based delegation

Example: Square Class (the right way)

```java
// Override setLength and setWidth
public void setLength(double l) {
    // could throw an exception
}

public void setWidth(double l) {
    // could throw an exception
}

public void setSide(double s){
    setLength(s);
    setWidth(s);
}
```

```java
public class Square1 {
    private Rectangle rectangle;

    public Square1() {
        rectangle = new Rectangle();
    }

    public Square1(double s) {
        rectangle = new Rectangle(s, s);
    }
}
```
Square Class (the right way)

```java
public void setSide(double s) {
    rectangle.setLength(s);
    rectangle.setWidth(s);
}

public double area() {
    return rectangle.area();
}
```

Multiple Inheritance

- We can use delegation to implement multiple class inheritance if necessary
- For instance:
  - instead of this:
  - you can do this:

Multiple Inheritance Example

```java
interface Student {
    public float getGPA();
}
interface Employee {
    public float getSalary();
}
public class StudentImpl implements Student {
    protected float GPA;
    public float getGPA() {
        // code for GPA
    }
}
```

Multiple Inheritance Example (continued)

```java
public class EmployeeImpl implements Employee {
    protected float salary;
    public float getSalary() {
        // code for Salary
    }
}
```

```java
public class StudentEmployee implements Student, Employee {
    private StudentImpl studentImpl;
    private EmployeeImpl employeeImpl;
}
Multiple Inheritance Example (continued)

```java
public StudentEmployee() {
    studentImpl = new StudentImpl();
    employeeImpl = new EmployeeImpl();
}
public float getGPA() {
    return studentImpl.getGPA();
}
public float getSalary() {
    return employeeImpl.getSalary();
}
```

Marker Interfaces

- Interfaces with no method or constant declarations
- Used to establish a subtype that is intended to have certain properties
- Most popular marker interface in Java: `Cloneable`
  - Distinguishes classes that can be cloned from those that cannot
  - Does not mean `clone()` method will succeed!
- Use them with caution.

Name Collisions Among Interfaces

- A java class may extend another class and implement one or more interfaces
- Inherited method from one interface may have same name as a method in another class or interface
- Name Collision procedure:
  - if methods have different signatures
  - if they have same signature and return type,
  - if they have same signature, different return types,
  - if they have same signature and return type, but throw different exceptions,

General Design Guidelines for Inheritance

- Place common attributes and methods in the superclasses
- Use inheritance to model only is-a type relationships
- Use polymorphism, not run-time type tests
  - Avoid code of the form
    ```java
    if (x instanceof class1)
        . . .
    else if (x instanceof class2)
        . . .
    ```
General Design Guidelines for Inheritance

- Use abstract classes and interfaces to design extensible families of objects with common properties
  - i.e. employees of different types
  - i.e. different type of students