Today

- Code review and (re)design of an MVC application
- OO design principles
  - Information hiding (and encapsulation)
  - Polymorphism
  - Open/closed principle
  - Inheritance in Java
  - The diamond of death
  - Liskov substitution principle
  - Composition/aggregation over inheritance

Let’s review the code of the following application

Source code available on the course web site

Information hiding

```
public class MyClass {
    public int nElem;
    public int capacity;
    public int top;
    public int[] elems;
    public boolean canResize;
    ...
    public void resize(int s){...}
    public void push(int e){...}
    public int capacityLeft(){...}
    public int getNumElem(){...}
    public int pop(){...}
    public int[] getElems(){...}
}
```

What does MyClass do?
Information hiding

Stack
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]

public class Stack {
    public int nElem;
    public int capacity;
    public int top;
    public int[] elems;
    public boolean canResize;
    ...
    public void resize(int s){...}
    public void push(int e){...}
    public int capacityLeft(){...}
    public int getNumElem(){...}
    public int pop(){...}
    public int[] getElems(){...}
}

Anything that could be improved in this implementation?

Information hiding

Stack
+ nElem : int
+ capacity : int
+ top : int
+ elems : int[]
+ canResize : bool
+ resize(s:int):void
+ push(e:int):void
+ capacityLeft():int
+ getNumElem():int
+ pop():int
+ getElems():int[]

Information hiding:
- Reveal as little information about internals as possible.
- Separate public interface from implementation details.
- Reduce complexity.

Information hiding vs. visibility

Public

???

Private

Public

???

Private

OO design principles
- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

A little refresher: what is Polymorphism?
A little refresher: what is Polymorphism?
An object’s ability to provide different behaviors.

Types of polymorphism
- Ad-hoc polymorphism (e.g., operator overloading)
  - \( a + b \Rightarrow \text{String vs. int, double, etc.} \)
- Subtype polymorphism (e.g., method overriding)
  - \( \text{Object obj = ...;} \Rightarrow \text{toString()} \) can be overridden in subclasses
    \( \text{obj.toString();} \) and therefore provide a different behavior.
- Parametric polymorphism (e.g., Java generics)
  - \( \text{class LinkedList}\langle E \rangle \) \( \Rightarrow \text{A LinkedList can store elements regardless of their type but still} \)
    \( \text{E get(int index)} \) \( \Rightarrow \text{provide full type safety.} \)

OO design principles
- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

Open/closed principle
Software entities (classes, components, etc.) should be:
- open for extensions
- closed for modifications

\[
\begin{align*}
\text{public static void draw(Object o)} & \{ \\
& \quad \text{if (o instanceof Square)} \{ \\
& \quad \quad \text{drawSquare((Square) o)} \} \\
& \quad \text{else if (o instanceof Circle)} \{ \\
& \quad \quad \text{drawCircle((Circle) o)} \} \\
& \quad \text{else} \{ \\
& \quad \quad \ldots \} \\
& \}
\end{align*}
\]

Violates the open/closed principle!

\[
\begin{align*}
\text{public static void draw(Shape s)} & \{ \\
& \quad \text{if (s instanceof Shape)} \{ \\
& \quad \quad \text{s.draw();} \\
& \quad \} \\
& \quad \text{else} \{ \\
& \quad \quad \ldots \} \\
& \}
\end{align*}
\]

Subtype polymorphism is essential to many OO design principles.

Open/closed principle
Software entities (classes, components, etc.) should be:
- open for extensions
- closed for modifications

\[
\begin{align*}
\text{public static void draw(Object o)} & \{ \\
& \quad \text{if (o instanceof Square)} \{ \\
& \quad \quad \text{drawSquare((Square) o)} \} \\
& \quad \text{else if (o instanceof Circle)} \{ \\
& \quad \quad \text{drawCircle((Circle) o)} \} \\
& \quad \text{else} \{ \\
& \quad \quad \ldots \} \\
& \}
\end{align*}
\]

Good or bad design?

\[
\begin{align*}
\text{public static void draw(Shape s)} & \{ \\
& \quad \text{if (s instanceof Shape)} \{ \\
& \quad \quad \text{s.draw();} \\
& \quad \} \\
& \quad \text{else} \{ \\
& \quad \quad \ldots \} \\
& \}
\end{align*}
\]

Violates the open/closed principle!
OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

Inheritance: (abstract) classes and interfaces

**LinkedList** extends **SequentialList**

**SequentialList** (abstract)

**LinkedList**

Inheritance: (abstract) classes and interfaces

**LinkedList** extends **SequentialList**

**SequentialList** (abstract)

**LinkedList**

Inheritance: (abstract) classes and interfaces

**LinkedList** extends **SequentialList** implements **List**, **Deque**

**SequentialList** (abstract)

**List**

**Deque**

**LinkedList**

**Iterable**

**Collection**
Inheritance: (abstract) classes and interfaces

```
<<interface>> Iterable
<table>
<thead>
<tr>
<th>extends</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;interface&gt;&gt; List</td>
</tr>
<tr>
<td>extends</td>
</tr>
<tr>
<td>&lt;&lt;interface&gt;&gt; Collection</td>
</tr>
</tbody>
</table>
```

List extends Iterable, Collection

OO design principles

- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

The “diamond of death”: the problem

```
A getNum():int

A a = new D();
int num = a.getNum();
```

Which getNum() method should be called?

The “diamond of death”: concrete example

Can this happen in Java? Yes, with default methods in Java 8.
**Design principles: Liskov substitution principle**

**Motivating example**

We know that a square is a special kind of a rectangle. So, which of the following OO designs makes sense?

- Square
- Rectangle

**Subtype requirement**

Let object $x$ be of type $T_1$ and object $y$ be of type $T_2$. Further, let $T_2$ be a subtype of $T_1$ ($T_2 < T_1$). Any provable property about objects of type $T_1$ should be true for objects of type $T_2$.

```java
Rectangle r = new Rectangle(2,2);
new Square(2);
int A = r.getArea();
int w = r.getWidth();
r.setWidth(w * 2);
assertEquals(A * 2, r.getArea());
```

Violates the Liskov substitution principle!
Design principles: Liskov substitution principle

Subtype requirement
Let object x be of type T1 and object y be of type T2. Further, let T2 be a subtype of T1 (T2 <: T1). Any provable property about objects of type T1 should be true for objects of type T2.

Rectangle
+ width : int
+ height : int
+ setWidth(w : int)
+ setHeight(h : int)
+ getArea() : int

<interface>
Shape

Rectangle

Square

OO design principles
- Information hiding (and encapsulation)
- Polymorphism
- Open/closed principle
- Inheritance in Java
- The diamond of death
- Liskov substitution principle
- Composition/aggregation over inheritance

Design choice: inheritance or composition?

Pros
- No delegation methods required.
- Reuse of common state and behavior.

Cons
- Exposure of all inherited methods (a client might rely on this particular superclass -> can't change it later).
- Changes in superclass are likely to break subclasses.

Composition/aggregation over inheritance allows more flexibility.

Inheritance vs. (Aggregation vs. Composition)

Hmm, both designs seem valid -- what are pros and cons?

Pros
- Highly flexible and configurable: no additional subclasses required for different compositions.

Cons
- All interface methods need to be implemented -> delegation methods required, even for code reuse.

Design choice: inheritance or composition?

List
- List
  - List
    - LinkedList
      - Stack
        - Stack
          - Stack
            - Stack
              - Stack
                - Stack

OO design principles: summary
- Information hiding (and encapsulation)
- Open/closed principle
- Liskov substitution principle
- Composition/aggregation over inheritance