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

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
**Information hiding**

**MyClass**

- 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[]

```java
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(){...}
}
```

**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[]

```java
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(){...}
}
```

**What does MyClass do?**

**Anything that could be improved in this implementation?**

**Information hiding:**

- Reveal as little information about internals as possible.
- Segregate public interface and implementation details.
- Reduces complexity.

```java
- elems : int[]
...```

**Information hiding:**

- Reveal as little information about internals as possible.
- Segregate public interface and implementation details.
- Reduces complexity.
### Information hiding vs. visibility

<table>
<thead>
<tr>
<th>Public</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Protected, package-private, or friend-accessible (C++).</td>
</tr>
<tr>
<td></td>
<td>Not part of the public API.</td>
</tr>
<tr>
<td></td>
<td>Implementation detail that a subclass/friend may rely on.</td>
</tr>
</tbody>
</table>

### 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)
  - $a + b \Rightarrow$ String vs. int, double, etc.

- Subtype polymorphism (e.g., method overriding)
  - `Object obj = ...;` ⇒ `toString()` can be overridden in subclasses and therefore provide a different behavior.

- Parametric polymorphism (e.g., Java generics)
  - `class LinkedList<E> {` ⇒ A LinkedList can store elements regardless of their type but still provide full type safety.
    - `void add(E) {...}`
    - `E get(int index) {...}`

Subtype polymorphism is essential to many OO design principles.

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

```java
public static void draw(Object o) {
    if (o instanceof Square) {
        drawSquare((Square) o);
    } else if (o instanceof Circle) {
        drawCircle((Circle) o);
    } else {
        ...
    }
}
```

**Good or bad design?**

<table>
<thead>
<tr>
<th>Square</th>
<th>+ drawSquare()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>+ drawCircle()</td>
</tr>
</tbody>
</table>
Open/closed principle

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

```java
public static void draw(Object o) {
    if (o instanceof Square) {
        drawSquare((Square) o);
    } else if (o instanceof Circle) {
        drawCircle((Circle) o);
    } else {
        ...
    }
}
```

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}

extends

LinkedList

Inheritance: (abstract) classes and interfaces

**LinkedList extends SequentialList**

SequentialList {abstract}

extends

LinkedList

Inheritance: (abstract) classes and interfaces

**LinkedList extends SequentialList implements List, Deque**

SequentialList {abstract}

extends

LinkedList

implements

List, Deque

Inheritance: (abstract) classes and interfaces

**<interface>> List**

**<interface>> Deque**

Inheritance: (abstract) classes and interfaces

**<interface>> List**

**<interface>> Iterable**

**<interface>> Collection**
Inheritance: (abstract) classes and interfaces

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 a = new D();
int num = a.getNum();
...
The “diamond of death”: the problem

```java
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?*

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

**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 $T_2$.

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ width :int</td>
</tr>
<tr>
<td>+ height:int</td>
</tr>
<tr>
<td>+ setWidth(w:int)</td>
</tr>
<tr>
<td>+ setHeight(h:int)</td>
</tr>
<tr>
<td>+ getArea():int</td>
</tr>
</tbody>
</table>

Is the subtype requirement fulfilled?

```
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 \( T_1 \) 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 \( T_2 \).

**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?**

Inheritance vs. (Aggregation vs. Composition)

<table>
<thead>
<tr>
<th>Person</th>
<th>Student</th>
</tr>
</thead>
</table>
| Public class Student extends Person{ public Student(){
 | public Bank(Customer c){
 | this.c = c;
 | }
 | ... } |

is-a relationship

<table>
<thead>
<tr>
<th>Customer</th>
<th>Bank</th>
</tr>
</thead>
</table>
| Public class Bank { Customer c; public Bank(Customer c){ this.c = c; } }
| ... |

has-a relationship

<table>
<thead>
<tr>
<th>Room</th>
<th>Building</th>
</tr>
</thead>
</table>
| Public class Building { Room r; public Building(){ this.r = new Room(); } }
| ... |

Design choice: inheritance or composition?

<table>
<thead>
<tr>
<th>Linkedlist</th>
<th>Stack</th>
</tr>
</thead>
</table>
| Public class LinkedList implements List{ private List l = new LinkedList(); }
| ... |

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

OO design principles: summary

- Information hiding (and encapsulation)
- Open/closed principle
- Liskov substitution principle
- Composition/aggregation over inheritance