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(){...}
}
Information hiding

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

Anything that could be improved in this implementation?

Information hiding

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

Information hiding vs. visibility

Public

???

Private

A little refresher: what is Polymorphism?

- 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?

An object’s ability to provide different behaviors.

Types of polymorphism

- Ad-hoc polymorphism (e.g., operator overloading)
  - $a + b$ ⇒ String vs. int, double, etc.

- Subtype polymorphism (e.g., method overriding)
  - `Object obj = ...;` ⇒ `toString()` can be overridden in subclasses
    - `obj.toString();` 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
    - `E get(int index) { ... }` provide full type safety.

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

```
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?

```
Square
+ drawSquare()

Circle
+ drawCircle()
```

Violates the open/closed principle!

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

```
<<interface>>
Shape
+ draw()

Square
Circle
...
```


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**

**SequentialList** (abstract)  
`extends`  
LinkedList

Inheritance: (abstract) classes and interfaces

**LinkedList extends SequentialList implements List, Deque**

**SequentialList** (abstract)  
`extends`  
LinkedList

Inheritance: (abstract) classes and interfaces

**LinkedList extends SequentialList implements List, Deque**

**SequentialList** (abstract)  
`extends`  
LinkedList

Inheritance: (abstract) classes and interfaces

**LinkedList extends SequentialList implements List, Deque**

**SequentialList** (abstract)  
`extends`  
LinkedList

Inheritance: (abstract) classes and interfaces

**LinkedList extends SequentialList implements List, Deque**

**SequentialList** (abstract)  
`extends`  
LinkedList
Inheritance: (abstract) classes and interfaces

```
<<interface>>  Iterable
extends
<<interface>>  Collection
extends
<<interface>>  List
```

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();
...  
```

Which getNum() method should be called?

The “diamond of death”: concrete example

Can this happen in Java? Yes, with default methods in Java 8.
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 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?

![Diagram showing the relationship between Rectangle and Square]

Is the subtype requirement fulfilled?

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 \( T_1 \) (\( T_2 \subseteq T_1 \)). Any provable property about objects of type \( T_1 \) should be true for objects of type \( T_2 \).

```
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

---

### Inheritance vs. (Aggregation vs. Composition)

**Inheritance**

- Person
  - Student

**Aggregation**

- Customer
  - Bank
  - Building

- Room

**Composition**

- Customer
  - Bank: `Customer c;`
  - Building: `public class Building { Room r; public Building() { r = new Room(); }...;`  

**Problems:**

- Is-a relationship vs. has-a relationship

---

### Design choice: inheritance or composition?

**List**

- `List` <<interface>>

- `LinkedList`  

- `Stack`  

- **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 and can’t change it later).
  - Changes in superclass are likely to break subclasses.

**List**

- `List` <<interface>>

- `LinkedList`  

- **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.

**Composition/aggregation over inheritance allows more flexibility.**

---

### Design choice: inheritance or composition?

**List**

- `List` <<interface>>

- `LinkedList`  

- `Stack`  

**Pros**

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