Recap
- Design principles
  - Separation of concerns
  - Information hiding (and encapsulation)
  - Composition/aggregation over inheritance
  - Open/closed principle
  - Liskov substitution principle

Today
- Homework 1: scope and questions.
- Design problems & potential solutions.
- Design patterns
  - What is a design pattern?
  - Categories of design patterns.

A little refresher
So wait, how can I tell the difference between inheritance, aggregation, and composition?

Polymorphism means what again and why is it important?

Why do I care about this composed view thing? What actual problem does it solve?
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 $obj = ...$;  \Rightarrow $toString()$ can be overridden in subclasses $obj.toString()$ and therefore provide a different behavior.

- **Parametric polymorphism** (e.g., Java generics)
  - class LinkedList<$E$>  \Rightarrow The list can store elements regardless of their type but still $E get(int index)$ provide full type safety.

We are focusing on subtype polymorphism.

What about class and students?
What about body and body parts?
Why use Polymorphism?

```java
public Integer getMin(LinkedList<Integer> list) {
    sort(list);
    return list.get(0);
}
```

```java
public void sort(LinkedList<Integer> list) {
    ... // sort the list
}
```

What if we want to use an ArrayList and a LinkedList?

```java
List<Integer> list1 = new LinkedList<>();
List<Integer> list2 = new ArrayList<>();
... // add elements to the lists
Integer min1 = getMin(list1);
Integer min2 = getMin(list2);
System.out.println("Min list1: "+ min1);
System.out.println("Min list2: "+ min2);
}
```

We can solve these problems with subtype polymorphism.

```java
public Integer getMin(List<Integer> list) {
    sort(list);
    return list.get(0);
}
```

```java
public void sort(List<Integer> list) {
    ... // sort the list
}
```
A first design problem

Weather station revisited

<table>
<thead>
<tr>
<th>Current</th>
<th>30 day history</th>
</tr>
</thead>
<tbody>
<tr>
<td>25° F</td>
<td>min: 20° F max: 35° F</td>
</tr>
<tr>
<td>-3.9° C</td>
<td></td>
</tr>
</tbody>
</table>

What's a good design for the view?

Weather station: view

```
public void draw(Data d) {
  for (View v : views) {
    v.draw(d);
  }
}
```

Weather station: view

```
class ComplexView {
  List<View> views;
  void draw(Data d) {
    for (View v : views) {
      v.draw(d);
    }
  }
}
```

ComplexView
<<interface>>
View
+draw(d:Data)
SimpleView GraphView ... F
-3.9° C min: 20° F
max: 35° F
+draw(d:Data)+draw(d:Data)
How do we need to implement draw(d:Data)?

Client sees
uses
manipulates
updates
09/01, 12°
09/02, 14°...
...View
25° F
-3.9° C
min: 20° F
max: 35° F
+draw(d:Data)
Design pattern: Composite

Composite
<<interface>>
Component
+operation()

CompA
+operation()

CompB
+operation()
+operation()

-comps:Collection<Component>
+operation()
+addComp(c:Component)
+removeComp(c:Component)

1..n
Iterate over all composed components (comps), call operation() on each, and potentially aggregate the results.

What is a design pattern?

- Addresses a recurring, common design problem.
- Provides a generalizable solution.
- Provides a common terminology.

Pros
- Improves communication and documentation.
- “Toolbox” for novice developers.
What is a design pattern?

- Addresses a recurring, common design problem.
- Provides a generalizable solution.
- Provides a common terminology.

Pros
- Improves communication and documentation.
- “Toolbox” for novice developers.

Cons
- Risk of over-engineering.
- Potential impact on system performance.

More than just a name for common sense and best practices.

Design patterns: categories

1. Structural
   - Composite
   - Decorator
   - ...
2. Behavioral
   - Template method
   - Visitor
   - ...
3. Creational
   - Singleton
   - Factory (method)
   - ...

1. Structural
   - Composite
   - Decorator
   - ...
2. Behavioral
   - Template method
   - Visitor
   - ...
3. Creational
   - Singleton
   - Factory (method)
   - ...
Another design problem: I/O streams

FileInputStream
+read():int
+read(buf:byte[]):int

InputStream
+read():int
+read(buf:byte[]):int

Problem: filesystem I/O is expensive

Solution: use a buffer!

Why not simply implement the buffering in the client or subclass?

Still returns one byte (int) at a time, but from its buffer, which is filled by calling read(buf:byte[]).
Design pattern: Decorator

Composite vs. Decorator