Recap

- Modeling and abstraction
- Software architecture vs. software design
Recap: software architecture examples

- **Pipe and filter**

  ```
  grep CS520 grades.csv | cut -f 1 -d ',' | sort | uniq -c
  ```

- **N-tier / client-server**

- **MVC (Model-View-Controller)**
Recap: software architecture principles

- Separation of concerns
- High-level encapsulation
- Manageable complexity
- Well-defined public API
Today

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

- A first design problem (if time permits)
Information hiding

<table>
<thead>
<tr>
<th>MyClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nElem : int</td>
</tr>
<tr>
<td>+ capacity : int</td>
</tr>
<tr>
<td>+ top : int</td>
</tr>
<tr>
<td>+ elems : int[]</td>
</tr>
<tr>
<td>+ canResize : bool</td>
</tr>
<tr>
<td>+ resize(s:int):void</td>
</tr>
<tr>
<td>+ push(e:int):void</td>
</tr>
<tr>
<td>+ capacityLeft():int</td>
</tr>
<tr>
<td>+ getNumElem():int</td>
</tr>
<tr>
<td>+ pop():int</td>
</tr>
<tr>
<td>+ getElems():int[]</td>
</tr>
</tbody>
</table>

What does this class do?
## Information hiding

### Stack

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nElem : int</td>
<td></td>
</tr>
<tr>
<td>+ capacity : int</td>
<td></td>
</tr>
<tr>
<td>+ top : int</td>
<td></td>
</tr>
<tr>
<td>+ elems : int[]</td>
<td></td>
</tr>
<tr>
<td>+ canResize : bool</td>
<td></td>
</tr>
<tr>
<td>+ resize(s:int):void</td>
<td></td>
</tr>
<tr>
<td>+ push(e:int):void</td>
<td></td>
</tr>
<tr>
<td>+ capacityLeft():int</td>
<td></td>
</tr>
<tr>
<td>+ getNumElem():int</td>
<td></td>
</tr>
<tr>
<td>+ pop():int</td>
<td></td>
</tr>
<tr>
<td>+ getElems():int[]</td>
<td></td>
</tr>
</tbody>
</table>

What does this class do?

Anything wrong with this Stack implementation?
Information hiding

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nElem : int</td>
</tr>
<tr>
<td>+ capacity : int</td>
</tr>
<tr>
<td>+ top : int</td>
</tr>
<tr>
<td>+ elems : int[]</td>
</tr>
<tr>
<td>+ canResize : bool</td>
</tr>
<tr>
<td>+ resize(s:int):void</td>
</tr>
<tr>
<td>+ push(e:int):void</td>
</tr>
<tr>
<td>+ capacityLeft():int</td>
</tr>
<tr>
<td>+ getNumElem():int</td>
</tr>
<tr>
<td>+ pop():int</td>
</tr>
<tr>
<td>+ getElems():int[]</td>
</tr>
</tbody>
</table>

Information hiding:

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

Public

???

Private
Information hiding vs. visibility

- **Public**
- **Protected, package-private, or friend-accessible (C++)**
- Not part of the public API
- Implementation detail that a subclass/friend may rely on

- **Private**
Today

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

- A first design problem (if time permits)
Aggregation vs. composition

```
public class Stack<E> {
    private List<E> l;

    public Stack(...) {
    }
    ...
}
```

```
public class Stack<E> {
    private List<E> l;

    public Stack(...) {
    }
    ...
}
```
Aggregation vs. composition

Consider object ownership and lifetime to determine the correct type of association!
Design choice: inheritance or composition?

- **Inheritance**:
  ```java
  public class Stack<E> extends LinkedList<E> {
      private List<E> l = new LinkedList<>();
      ...
  }
  ```

- **Composition**:
  ```java
  public class Stack<E> {
      private List<E> l = new LinkedList<>();
      ...
  }
  ```
Design choice: inheritance or composition?

Think about *is-a* vs. *has-a* relationship first.
Design choice: inheritance or composition?

Which is the right choice?
Think about **is-a** vs. **has-a** relationship first.

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.

**Pros**
- Highly flexible and configurable.
- No additional subclasses required for different has-a relationships.

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

Composition/aggregation over inheritance allows more flexibility.
Classes, abstract classes, and interfaces

MyClass

public class MyClass {
    public void op() {
        ...
    }
    public int op2() {
        ...
    }
}

MyAbstractClass

{abstract}

public abstract class MyAbstractClass {
    public abstract void op();
    public int op2() {
        ...
    }
}

<<interface>>

MyInterface

public interface MyInterface {
    public void op();
    public int op2();
}

Recall how default methods (Java 8) fit into this spectrum?
Classes, abstract classes, and interfaces

- `List` (interface)
- `Deque` (interface)
- `Collection` (interface)
- `Iterable` (interface)

- `SequentialList` (abstract class)
- `LinkedList`

Extends:
- `SequentialList` extends `List`
- `LinkedList` implements `List` and `Iterable`
Composition/aggregation over inheritance

The “diamond of death”

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

```java
A +getNum():int

B +getNum():int

C +getNum():int

D
```
Composition/aggregation over inheritance

The “diamond of death”

\[
\begin{align*}
\ldots \\
A & \ a = \text{new } D(); \\
\text{int} & \ \text{num} = a.\text{getNum}(); \\
\ldots
\end{align*}
\]

Which version of getNum() should be called?

\[
\begin{align*}
A & \quad + \text{getNum():int} \\
B & \quad + \text{getNum():int} \\
C & \quad + \text{getNum():int} \\
D & \\
\end{align*}
\]
Composition/aggregation over inheritance

The “diamond of death”

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

Can you think of a particular method in Java for which this problem could arise (if Java would allow multiple inheritance)?
Today

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

- A first design problem (if time permits)
Design principles: open/closed principle

Software entities (classes, components, etc.) should be:

- **open** for extensions
- **closed** for modifications

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

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

Good or bad?
Design principles: open/closed principle

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

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

Violates the open/closed principle!
Design principles: open/closed principle

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

```java
public static void draw(Object f) {
    if (f instanceof Figure) {
        f.draw();
    } else {
        ...
    }
}
```

```java
public static void draw(Figure f) {
    f.draw();
}
```

---

```
<<interface>>>
Figure

+ draw()
```

- Square
- Circle
- ...

---
Today

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

- A first design problem (if time permits)
Motivating example

We know that a square is a special kind of a rectangle. So, which of the following OO designs makes sense?
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 \( T_1 \) \((T_2 \preceq T_1)\). 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?
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 \( T_2 \).

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

Rectangle r = new Rectangle(1, 2);
  r.setWidth(2);
  r.setHeight(10);
  int A = r.getArea();
  assertEquals(20, A);

Rectangle

Square
```
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 $T_2$.

```java
Rectangle r = new Rectangle(1,2);
new Square(2);
r.setWidth(2);
r.setHeight(10);
int A = r.getArea();
assertEquals(20, A);
```

Violates the Liskov substitution principle!
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></td>
</tr>
<tr>
<td>-3.9° C</td>
<td>min: 20° F</td>
</tr>
</tbody>
</table>

Reset history button

Temp. sensor
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></td>
</tr>
<tr>
<td>-3.9° C</td>
<td>min: 20° F max: 35° F</td>
</tr>
</tbody>
</table>

We focus on the view -> what would be a good design?
Weather station: view

```
<<interface>>
View
+ draw(d:Data)
```

- `SimpleView`:
  - `draw(d:Data)`

- `GraphView`:
  - `draw(d:Data)`

- `...View`:
  - `draw(d:Data)`

- `ComplexView`:
  - `draw(d:Data)`
  - `addView(v:View)`
Weather station: view

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