CSC301

Object-oriented topics: Composition vs. Inheritance, immutability & static factory methods

Object Oriented - The Basics

- We use classes & interfaces to model a *domain*
  - Classes and interfaces correspond to concepts in the domain (“the real world”)
- Some basic concepts:
  - Encapsulation - Objects hide their internals from other objects
  - Inheritance - Objects can acquire some/all of the properties of other objects
  - Polymorphism - Objects can behave differently based on their type.
    In other words, you switch between different implementations of the same interface

Object Oriented Programming

- ***Inheritance*** is a fundamental part of OO languages
  - Classes are organized into an inheritance hierarchy
  - A class inherits the implementation (methods & fields) of its parent
  - A class can override some of its parent's behaviour
- Inheritance is not the only relationship between classes, there is also ***composition***
  - Composition can be subcategorized further, but we won't get into it today
- *Inheritance* vs. *composition* ⇔ *"Is a"* vs. *"Uses/Has a"*
  - E.g.: "A dolphin *is a* mammal" vs. "A dolphin *has* lungs"

Let’s see a simple code example …

Inheritance

```
class BasicRobot {
    public void setAlarmClock(Time t){
        // ...
    }
}



class FancyRobot extends BasicRobot {
    public void makeCoffee(){
        // ...
    }
}
```

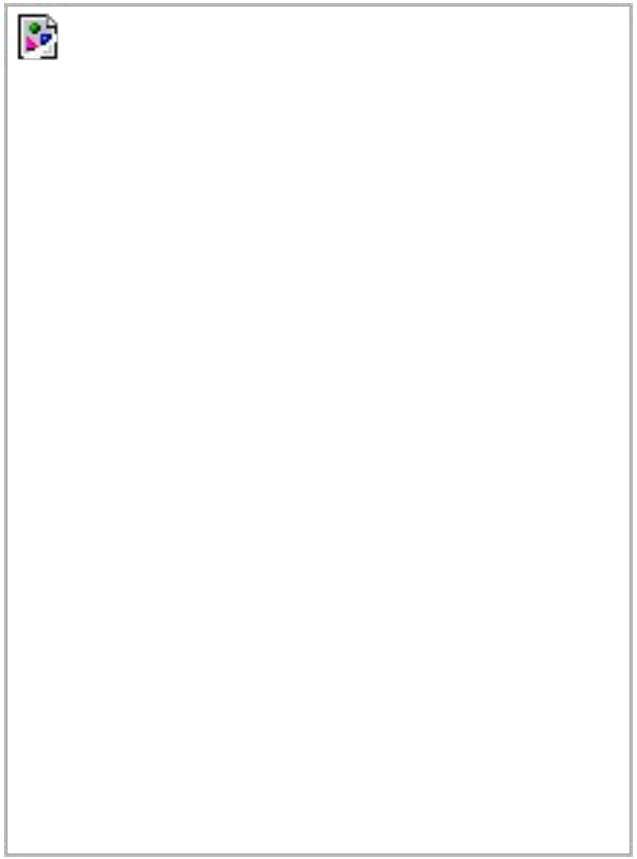
Composition

```
// BasicRobot is the same as before

class BasicRobot {
    public void setAlarmClock(Time t){
        // ...
    }
}
```

```
class FancyRobot {

    BasicRobot basicRobot;

    public FancyRobot(BasicRobot basicRobot){
        this.basicRobot = basicRobot;
    }

    public void setAlarmClock(Time t){
        this.basicRobot.setAlarmClock(t);
    }

    public void makeCoffee(){
        // ...
    }
}
```

Or, if you prefer UML diagrams over code ...

Inheritance



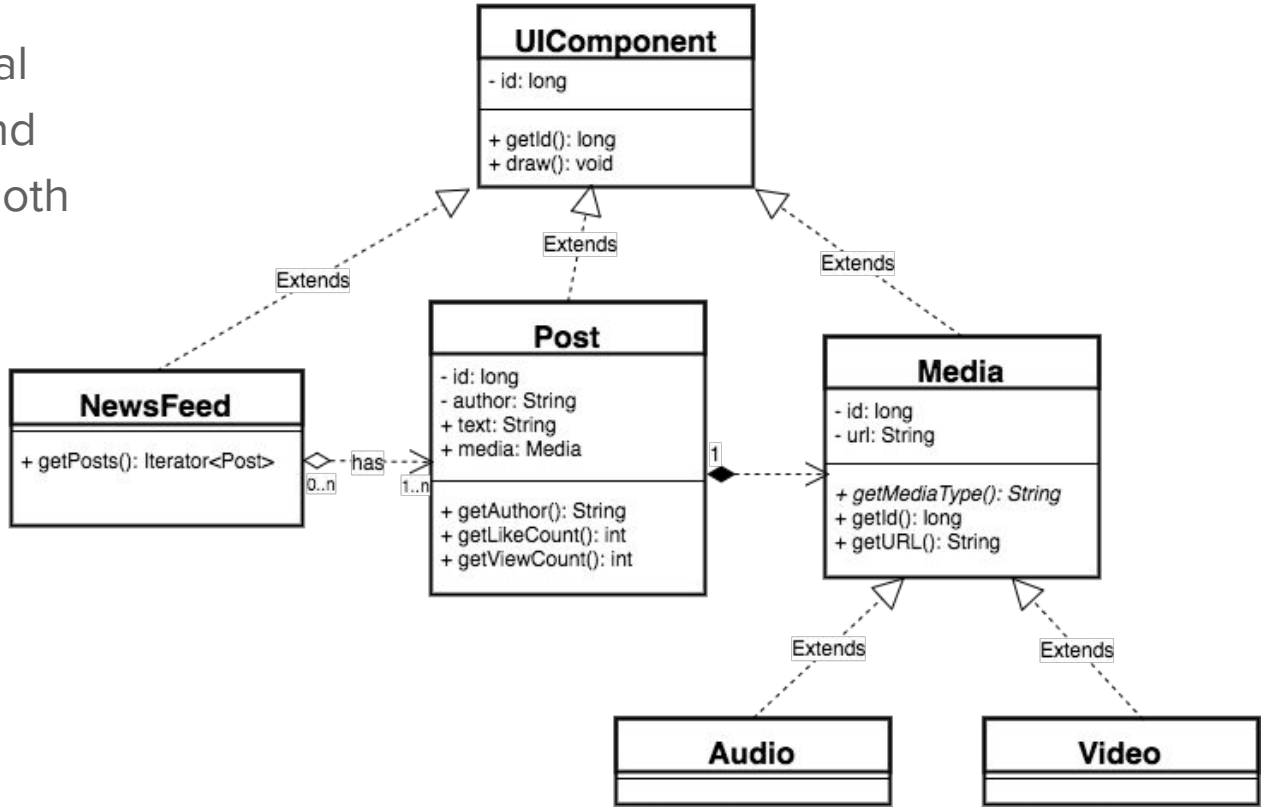
Composition

Inheritance vs. Composition

- Design decisions have implications.
  - Flexibility to respond to changing requirements
  - Clarity of the code
  - Ease of maintaining & extending the system over time
- So … which one should you choose? Inheritance or composition?
  - Depends on the domain you are modeling
  - In some cases composition will make more sense (e.g., a car has an engine, wheels, etc.), while in some other cases inheritance may feel more natural (e.g., biological classification/taxonomy).
  - This is a very nice blog post about the topic.

Inheritance vs. Composition

When building a real system, you can (and probably will) use both techniques.



Individual Assignment, A2

- Still dealing with robots
- You are asked to abstract the details of a low-level API, `IBasicRobot`, with a higher-level API, `IGridRobot`.
  - Feel free to reuse your `BasicRobot` implementation from A1
  - You will need to create two implementations for `IGridRobot`:
    - By composition with an `IBasicRobot`
    - By inheritance, extending your `BasicRobot` implementation

A2, Additional Notes

- Although the auto-marker doesn't check it, you should get used to using the proper access modifiers.
  - E.g.: Instance variables of `BasicRobot` should be private. Subclasses can get the x, y coordinates and/or rotation using getters.
- And let's talk about tests that use random values:
  - There is an obvious issue - You can test the same code and get different results.
    - **Important:** You are still expected to submit bug-free code!
      Otherwise, you are gambling … your code may or may not fail when the auto-marker runs.
  - That being said, there are some advantages:
    - Make the tests clear (specifies all possible valid values, for a given variable)
    - Make the tests short and easier to read

A2, Additional Notes

- If you have not accepted your Github invite and cannot access A2, email Adam Otherwise, you won't be able to access some of the course material later.
  - By not accepting the invite you are giving the impression that you (1) don't take the course as seriously as we expect you to (2) don't come to lectures, don't check the discussion board and ignore course-related emails.
- Next, let's take a closer look at some of the code in the assignment ...

```java
public class GridCell {
	public static GridCell at(int x, int y){
		return new GridCell(x, y);
	}

	public final int x;
	public final int y;

	private GridCell(int x, int y) {
		this.x = x;
		this.y = y;
	}

	@Override
	public boolean equals(Object other) {
		return other instanceof GridCell && this.x == ((GridCell) other).x && this.y == ((GridCell) other).y;
	}

	@Override
	public String toString() {
		return "[" + x + "," + y + "]";
	}
}
```

GridCell

- `GridCell` is a small convenience class that represent XY coordinates
  - In other words, a `GridCell` instance is just a pair of integers
  - You don’t need to change this class
    If you feel like you need to change it, then you are probably misunderstanding something about the starter code
  - General object-oriented concept - Keep classes small, simple and single purpose

GridCell

- If we look closely at the `GridCell` class, there are a few things we should mention
  - It is immutable - You cannot change the `x` or `y` coordinates of a GridCell instance
    Why do you think that's a good/bad thing?
  - It has a convenience *static factory method*
    E.g.: `GridCell.at(30, 1)`
  - Its constructor is private
    - Therefore, using the static factory method is the only way to create GridCell instances
    - Why is that a good idea?

GridCell

- What if our program calls `GridCell.at(x,y)` very frequently?
  - Every call creates a new `GridCell` instance
  - Might have many instances with identical `x` & `y` values (i.e., using memory)
  - Might make the garbage collector work really hard (i.e., using CPU)
- Can you think of an elegant solution?
  - **Important***:* You must stay backwards compatible! You cannot break existing code

GridCell

- Let’s take a TDD approach, and write the tests first ...
- First test (equality):
  - Create two GridCell instances with the same X & Y coordinates
  - Check that they are equal to one another
- Second test (same instance):
  - Create two GridCell instances with the same X & Y coordinates
  - Check that they are the *same instance*
  - This will fail right after passing the first test. We need to improve to make it pass by keeping a record of instances (e.g., caching)

GridCell - Improvements

- The improvement we just discussed (*caching* instances in memory, to avoid creating unnecessary objects) is
  - Probably unnecessary for the case of `GridCell`
  - Definitely not important in an early stage of a project.
- That being said, in some realistic cases, the same optimization technique can be very useful.
  - We want you to be able to apply it.
  - We want you to be able to understand what's going on, if/when you encounter it in a codebase you are working on.
  - You've probably seen *caching* in a different context in one of your first-year courses. Think about memoization and recursive functions (like computing the n^{th} Fibonacci number).