CS-230 Software Engineering

L06: Class Hierarchies, Abstract Classes and Interfaces

Dr. Liam O'Reilly

Semester 1 – 2020

Previously in CS 230...

`LibraryMember`	
Responsibilities	**Collaborators**
Maintain data about copies currently borrowed Meet requests to borrow and return copies	`Copy`

`Copy`	
Responsibilities	**Collaborators**
Maintain data about a particular copy of a book Inform corresponding `Book` when borrowed and returned	`Book`

`Book`	
Responsibilities	**Collaborators**
Maintain data about one book Know whether there are borrowable copies	

Live Up to Your Responsibilities!!!

Previously in CS 230...

- We talked about responsibilities.
- Responsibilities are...
  - Knowledge maintained by objects: attributes.
  - Actions a class can perform: behaviours/operations.
  - Should represent purpose of the class in system.
- What indicates a class?
  - The nouns in the specification indicate entities that might be good classes.
- What indicates a responsibility?
  - The verbs in active voice in the specification.

Previously in CS 230... (2)

- Properties of a good assignment of responsibilities is...
  - Distribute system intelligence evenly.
  - State responsibilities as generally as possible.
  - Keep behaviour (actions) with related information.
  - Keep information about one thing in one place.
  - Responsibilities can be shared among related objects.
  - Each class should have one main purpose, one idea, one main responsibility.

Shared Responsibilities

- We know about responsibilities.
- Sometimes responsibilities can be shared...
- But, duplication is our enemy...
- How can we deal with this?

Class Hierarchies

Class Hierarchies

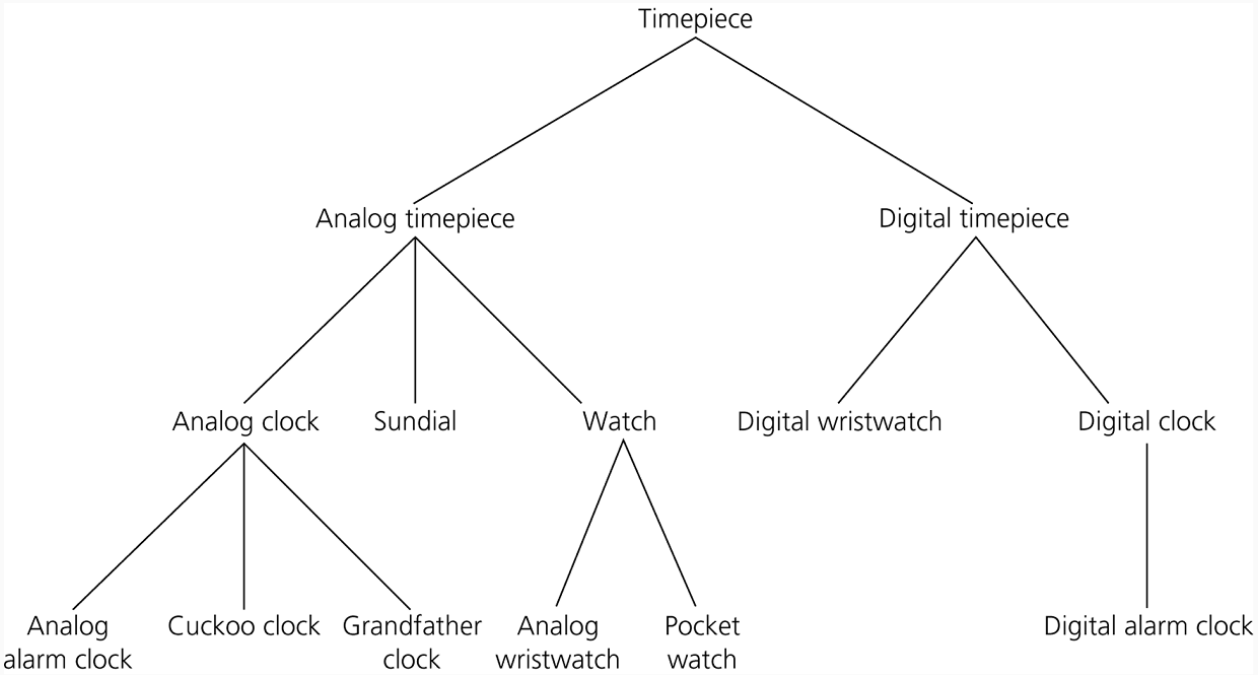
- Previous steps (classes, responsibilities) yield a preliminary design.
- To maximise the benefits of an object oriented approach we design class hierarchies.
  - Represent essence of Object Oriented approach.
  - Provide most potential benefits.
- Global view of design can help us understand and improve hierarchy.
- Inheritance helps us avoid repetition.

Inheritance

- Is-a relationship.
  - A mammal is an animal.
  - A bird is a animal.
  - A car is a vehicle.
  - An aeroplane is a vehicle.
- In these is-a relationships:
  - Animal and vehicle are superclasses: a general class type.
  - Mammal, bird, car, and aeroplane are subclasses: specific versions of a superclass.
- All subclasses are types of a superclass.
  - Put another way: subclasses are specialisations.

Example Inheritance Hierarchy

Example inheritance for types of clocks:



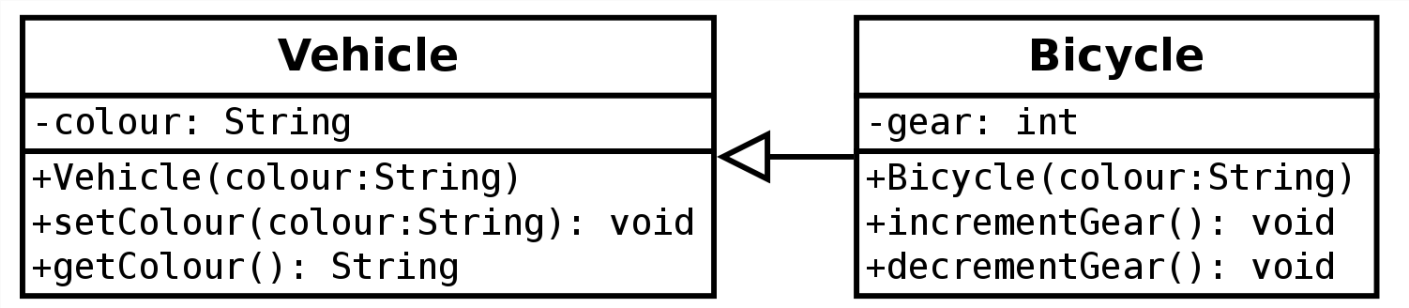
Superclass

- Provides attributes and operations common to all subclasses.
- Substitution Principal:
  - An instance of a subclass can be used when an instance of a superclass is expected.
  - Variables declared to be of a superclass type can actually store references to instances of subclasses.
  - Example. Say you have a method:
    `public void process(Timepiece t){...}`
    then we can call this method with an instance of a digital alarm clock (as `t`).
- Methods implement default behaviours (in superclass) unless overridden by subclass.
- Prevents re-implementation of common functionality in subclasses.

Subclass

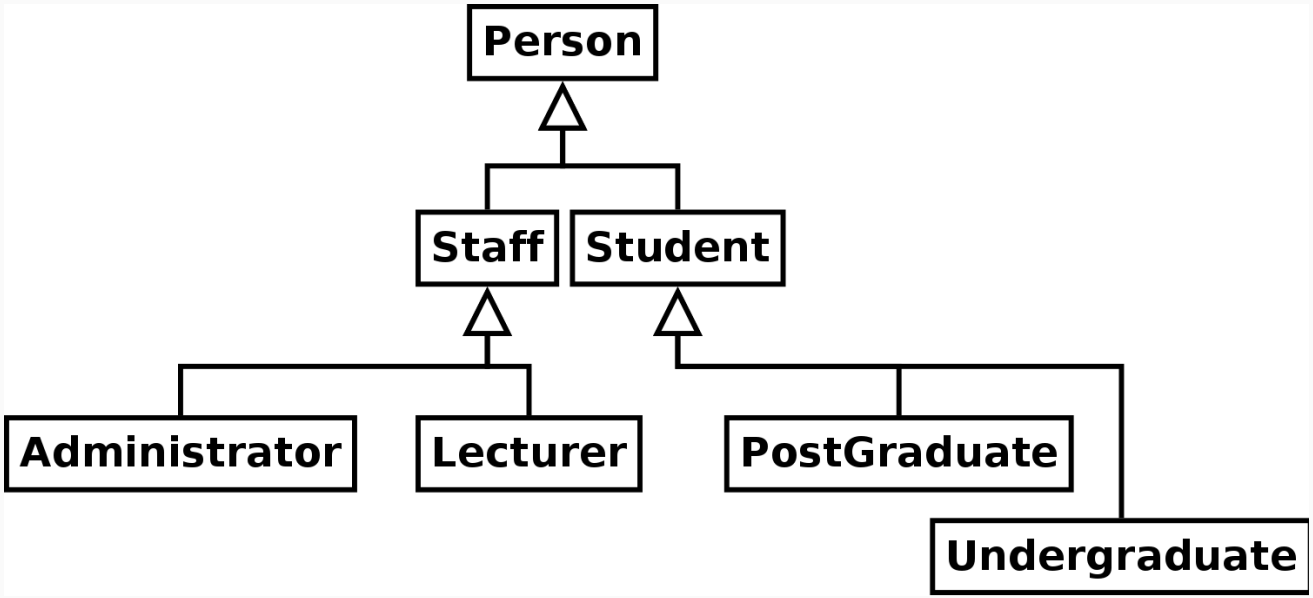
- A subclass is a more specific type of the superclass.
- Inherits the methods and attributes of the superclass.
- Additional methods and attributes can be added to the subclass.
- Methods in the superclass can be overridden:
    - Done by providing methods with the exact same name and parameters.

Example Inheritance in UML



- Simple UML hierarchy of a Vehicle and a Bicycle.
- A Bicycle can do everything a Vehicle can and more.
  - Do not repeat inherited attributes/operations in subclasses in Class Diagram.
- Hollow arrow always goes from subclass to superclass.
- In UML we use the words specialisation and generalisation instead of inheritance.

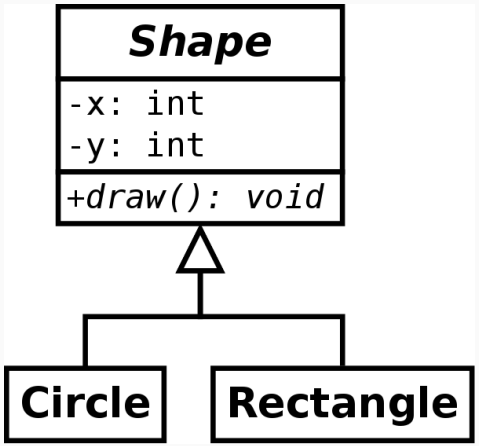
Complicated and Large UML Diagrams



- A model can have multiple diagrams:
  - Not all details must be present on each one.
  - Here we only show the hierarchy.
  - The details (attributes and operations) of the classes would need to be shown on other diagrams.
- Arrows can be individual or bunched (as above).

Abstract Classes and Operations

Abstract Methods and Classes



- Sometimes a method has different implementations based on the subclass.
- But you can not provide a good implementation in the superclass – the superclass is just too general!
- But you want to require the implementation of this method in all subclasses.
- E.g., How would you draw the most general shape? You can't.

Abstract Classes in Java

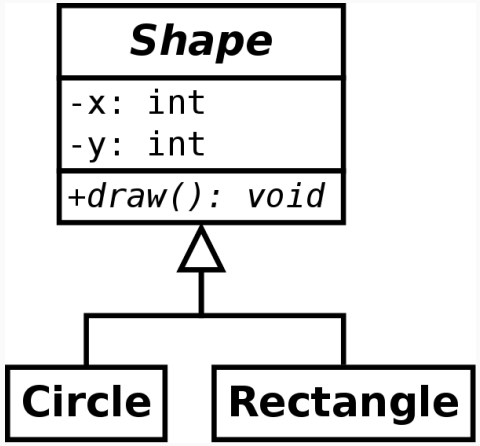
- We can declare a method in Java to be abstract.
- We provide no implementation (notice the semi colon).
- If a class contains an abstract method, then the class must be abstract too.
- Abstract classes can still contain normal methods.

```
public abstract class Shape {
  private int x;
  private int y;
  public abstract void draw();
}
```

The Point of Abstract Classes

- The compiler will not let us instantiate abstract classes.
  - Compile-time error.
- All subclass must either implement the abstract methods, or be abstract themselves.
- Classes with full implementations are known as concrete classes.
- This setup ensures concrete class always have the required methods. e.g., `draw`.
- Good Pratice: Abstract classes are never children of concrete ones.
- We can create variables of a type declared to be abstract. **Substitution principal** says the values are allowed to be references to concrete classes.

Abstract in UML Class Diagrams



- We represent abstract class in UML by making the class name italic.
- We represent abstract operations in UML by making the operation name italic.
- This italic style is the standard in UML, it is subtle and can be hard to see – be careful.

Designing Class Hierarchies

Designing Good Class Hierarchies

- Drawing out inheritance hierarchies can help in analysis of hierarchy.
- Guidelines:
  1. Model is-kind-of (or is-type-of) hierarchies.
  2. Factor common responsibilities as high as possible.
  3. Don't allow abstract classes to inherit from concrete ones.
  4. Eliminate non-functional classes.

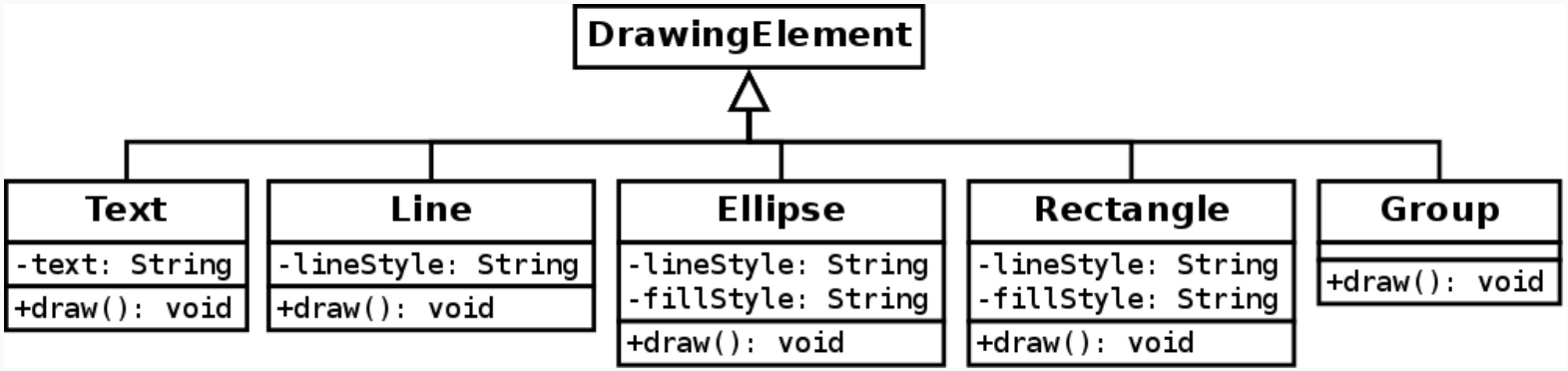
Move Responsibilities Up Hierarchy

- Move common responsibilities as high up as possible.
  - If many classes support a responsibility, they inherit it from a common parent.
- Make good use of abstract classes:
  - Likely result in simplification, design sharing, and code sharing.
  - Ease integration of future components (abstract methods).
  - Design of abstract classes can involve speculation on future extensions/modifications.
  - One shared responsibility is often enough to justify an abstract class.
  - Need at least two children (concrete uses), otherwise difficult to identify good generic definition of responsibilities.

Hierarchy Refinement Example

We want to design a system where the user can use lines, ellipses, rectangles, texts and groups of objects to make a drawing.

Our first idea:



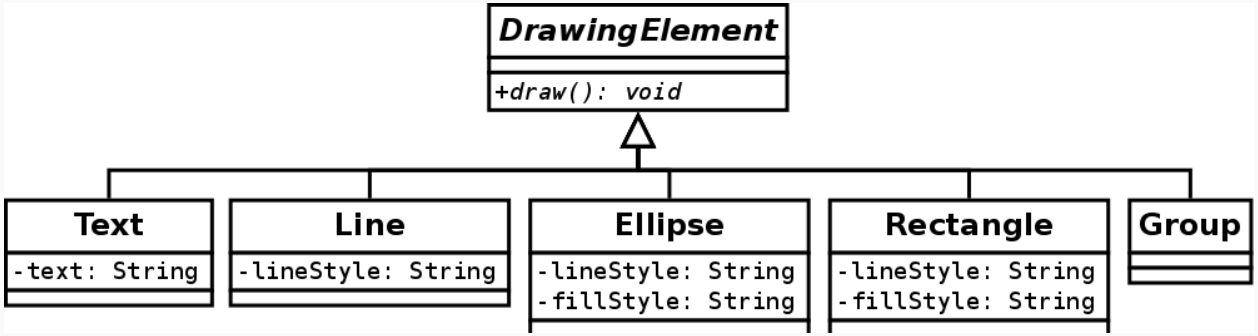
- Line style: solid, dashed, dotted.
- Fill style: filled, or transparent.
- We have ignored may details like colour, coordinates, etc.

Any problems here?

- Repeated attributes & repeated operations.

Hierarchy Refinement Example (1)

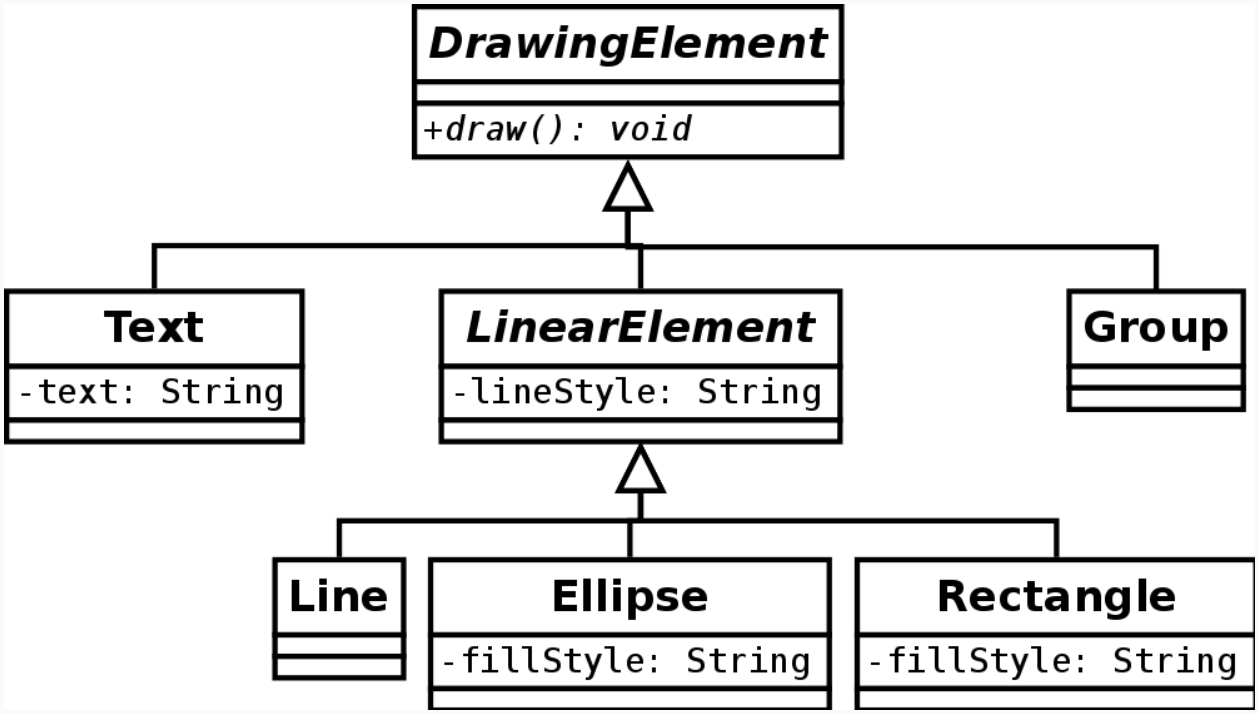
Let's get rid of the duplicate `draw` operations.



- Drawing Element is abstract because of abstract `draw` op:
  - We can't provide a good implementation of `draw`.
  - We want to enforce that all elements can draw themselves.
- `Line`, `Ellipse` and `Rectangle` all have a `lineStyle` and `fillStyle`.
  So why not move them up to `DrawingElement` too?
  - Responsibilities are not shared by `Text` and `Group`.
  - So it cannot be assigned to `DrawingElement`.

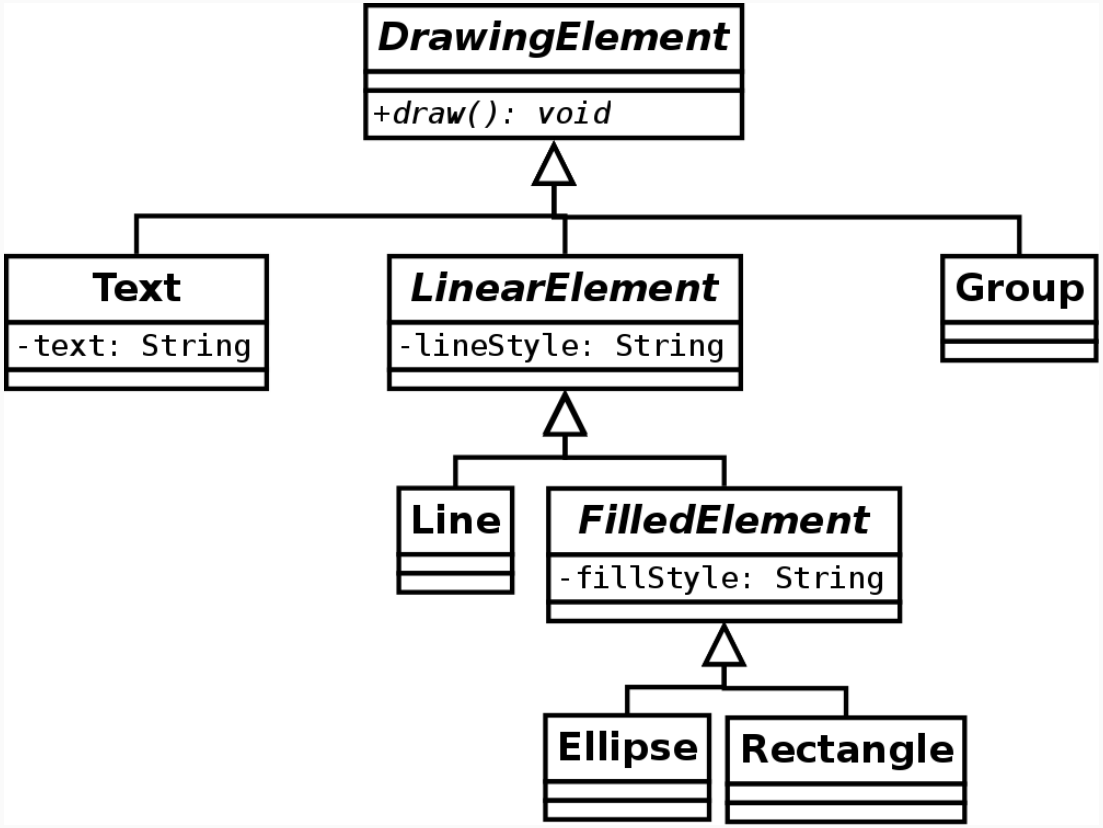
Hierarchy Refinement (2)

Let's get rid of the duplicate `lineStyle` attributes.



- The `lineStyle` problem is solved by abstract linear element.
- But what about `fillStyle`? `Line` Does not have this responsibility.

Hierarchy Refinement (3)



- We solve the `fillStyle` problem by adding `FilledElement`.
- Notice: `DrawingElement`, `LinearElement` and `FilledElement` are all abstract.
- We have used abstract a lot here, not everything you move up needs to be abstract!

Refine Your Hierarchy

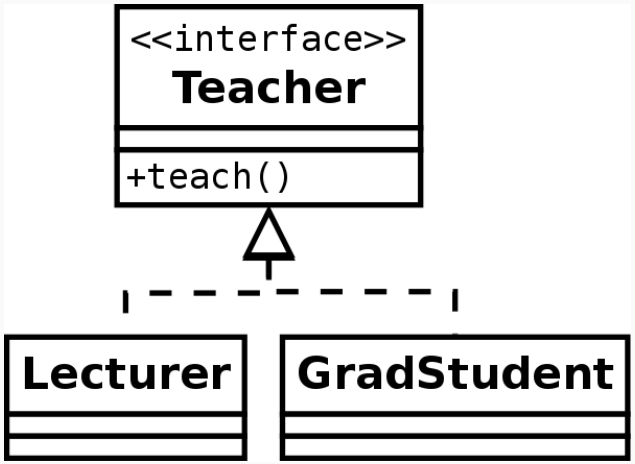
- Classes that add no new functionality are normally eliminated.
    - A class can have no new responsibilities, but still add functionality.
    - Responsibilities represent only public services.
    - Responsibilities are inherited by children, but assigned to parent.
    - But, if child implements the parent responsibility in a different way (e.g., Display methods, etc.), then it adds functionality.

Interfaces
(Not GUIs!)

Inheritance is Not Always The Answer

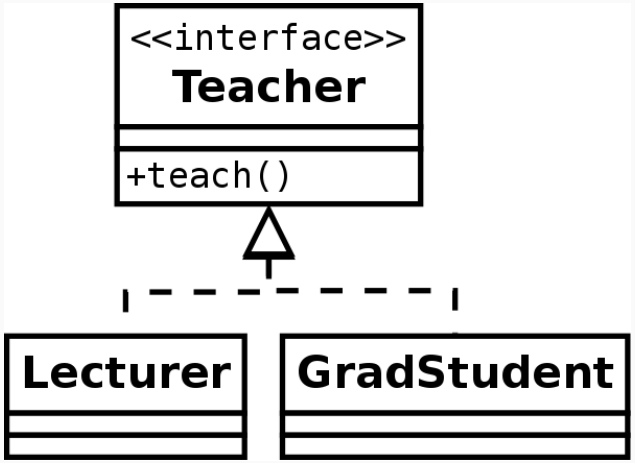
- Some time you cannot form inheritance hierarchies as the classes are not related naturally, but you still want to enforce that classes have a certain operations.
- Solution: Interfaces.
- Interfaces are just a set of public abstract method declarations.
- Interfaces cannot have attributes.
- Classes can implement multiple interfaces.
  - Keyword implements.
  - By a class declaring it implements an interface, it is stating is obeys the contract.
  - It must provide an implementation for each method in the interface.

Inheritance: UML Example



- Here, we have an interface `Teacher`. This is an interface, not a class as indicated by the text `<<interface>>`.
- It has one operation `teach`.
- There are two classes that each implement the interface (dashed arrows).
- These classes must provide and implement the method `teach`.

Inheritance: Java Example




```
public interface Teacher {
  public void teach();
}

public class Lecturer implements Teacher {
  public void teach() { ... }
}
```

Example code:

```
Teacher x = new Lecturer();
```

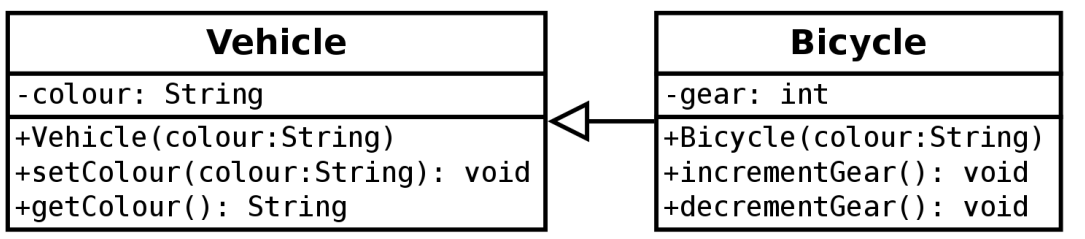
Quick Java Inheritance Recap

A Reminder

- We are now going to look at these concepts in Java.
- Remember, UML is not Java and design doesn't involve coding.
- Remember *software design is not coding*.
- This quick detour is to help you with implementation later on.

Inheritance in Java

- Keyword extends is used to specify inheritance in Java.




```
public class Vehicle {
  private String colour;
  public Vehicle(String color) {...}
  ...
}

public class Bicycle extends Vehicle {
  private int gear;
  public Bicycle(String color) {...}
  ...
}
```

What methods can be called?

- `Bicycle` cannot access any private methods or attributes of `Vehicle`.
- `Bicycle` can access any of its superclass' public or protected methods and attributes.
  - In UML, protected indicated with # (instead of + or −).
- If a method is overridden:
  - References to `Bicycle` objects use `Bicycle` method.
  - References to `Vehicle` objects use `Vehicle` method.
  - If a `Bicycle` reference is stored in a variable declared to hold `Vehicle` references, the `Bicycle` method is used
  - This is dynamic method dispatch and the substitution principal.

Short Review

1. What is a inheritance hierarchy?
2. What is a superclass?
3. What is a subclass?
4. What is the difference between abstract and concrete classes?
5. What is the purpose of an abstract class?