CompSci 230
Software Construction

Lecture 5: Object-Oriented Design, Part 2

Agenda & Reading

- Topics:
  - Abstraction and information hiding
  - Inheritance, instantiation, and polymorphism
  - Association, aggregation, and composition
- Reading
  - The Java Tutorials, on Inheritance
  - Wikipedia, on Class Diagram

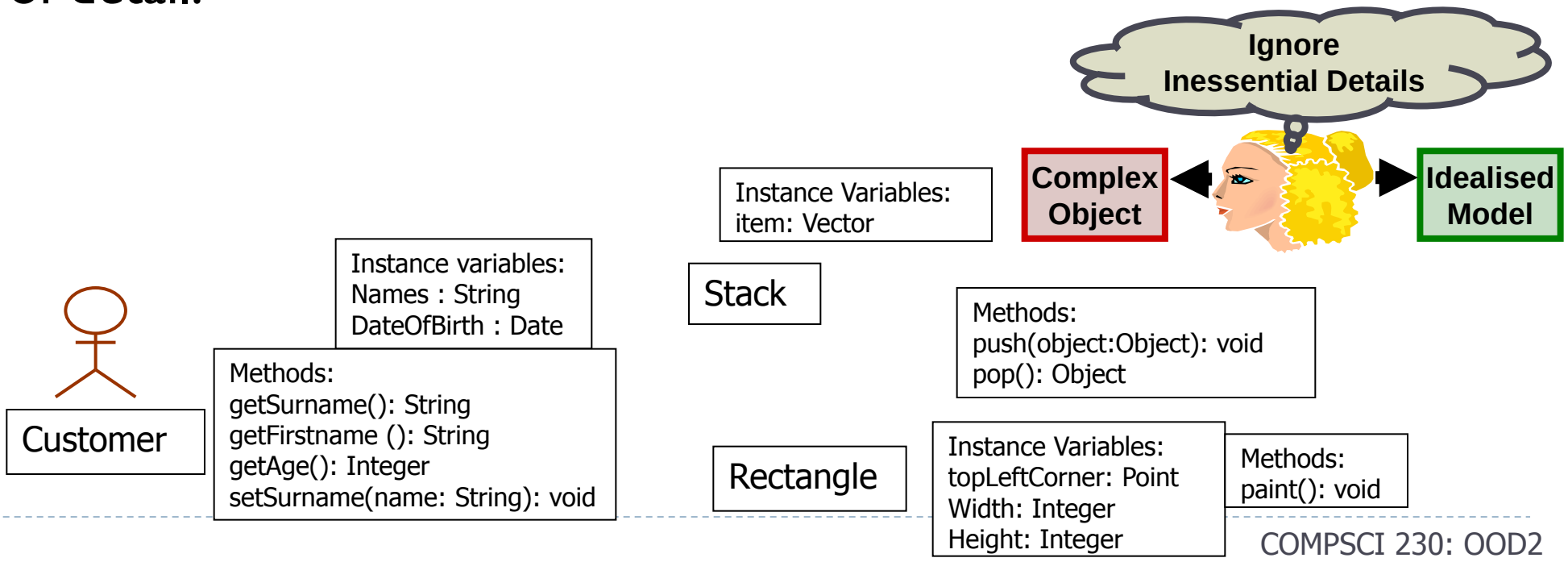
Learning Objectives

- Students will have a strong conceptual foundation for their future uses of the OO features of Java
  - Abstraction and information hiding (refined into two definitions)
  - Inheritance (**is-a**) vs composition/aggregation/association (**has-a**)
  - Polymorphism
- Students will be able to discuss the OO features of an existing design (as expressed in a Java program or a class diagram)
- Students will be competent at basic OO design

- Teaching strategy in this unit: first an overview, then "dig deeper" the second time around…

Abstraction

- An abstraction is a view or representation of an entity that includes **only** the attributes of significance in a particular context.
- Abstraction is essential when working with complex systems.
- Without abstraction, the programmer faces an overwhelming level of detail.



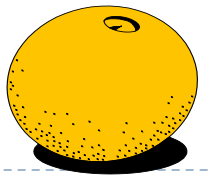
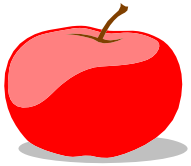
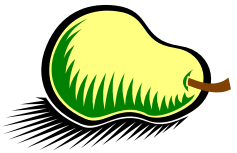
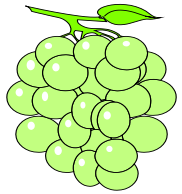
Information Hiding: two definitions

1. A programming technique. The programmer is trying to hide info by
   - using language features (such as `Interface`, `protected`, `private`) to restrict access to implementation details.
   - In the extreme case, other programmers are not allowed to "look inside" your library classes to see their implementations.
2. A design technique. The designer is trying to hide info by
   - defining a model which is as simple as possible (but useful, and unlikely to change).
   - In the extreme case, other programmers can read your high-level design documents, but they can not read (and need not read) any existing implementation.

- It is usually undesirable for programmers to rely on "undocumented functions" in an implementation, and type-1 info-hiding makes it harder for these to be discovered.
  - Undocumented functions are subject to change with every release
- Extreme type-2 info-hiding is usually undesirable
  - Design documents are rarely complete, accurate, and unambiguous.
  - Important requirements are often expressed only in the test-suite or in informal understandings among the devteam, the QA team, and primary stakeholders.

Inheritance and Instantiation

- **Inheritance**: create new classes from existing classes.
  - Instantiation: create new instances from existing classes.
  - Inheritance is more powerful than instantiation.
  - When a **subclass** is created by inheritance from a **superclass**, some of the methods and attributes in its superclass may be added or redefined.
- Inheritance is an "**is-a**" relation.
  - Example: "An orange is a fruit. An apple is a fruit."
    - Every instance of an Orange is a Fruit (= its superclass) , but it is more accurately described as being an instance of the Orange class (= its subclass).
    - If there is no important difference between oranges and apples, you should simplify your design!

Polymorphism

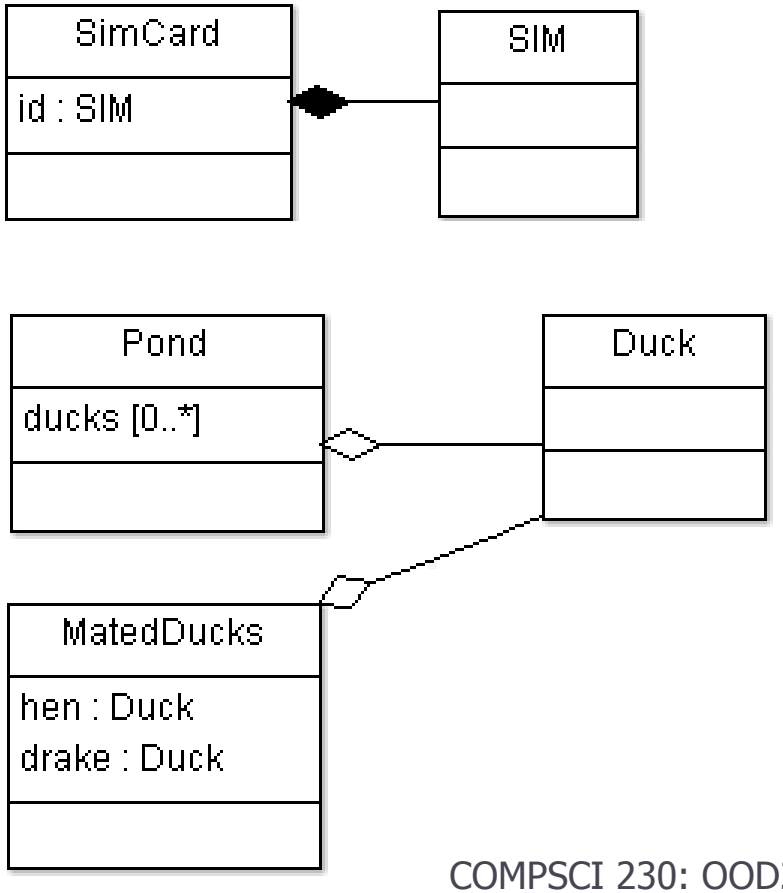
- Different objects can respond differently to the same message.
  - Inheritance is the "obvious" way to obtain polymorphic behaviour in your OO design, but it may not be the best way.
  - Instantiations are polymorphic, if the values in their attributes affect the behaviour of their methods.
- Hmmm… if you have instantiated a million objects of a single Class, could they do anything useful? Hmmm….
  - Worker ants are (nearly) identical, but they won't reproduce without a Queen ant.
  - Ants may be important members of an ecosystem, but only if the ecosystem contains other forms of life, some inanimate objects, and an energy source.
  - One way to conceive of OOD is that you're designing an ecosystem with multiple species (Classes) in an evolutionary tree.
  - It is possible to write a useful program in a non-OO language!
  - Polymorphism is not necessary in programming, but it is fundamental to OO design.

OO Analysis: Basic Questions

- What classes and instances should be in my high-level design?
  - To get started on answering this question, you should identify important entities, and look for commonalities among different entities.
    - Similar entities might be instances of the same class… but maybe there are some "natural" subclasses?
- How should my classes and instances be related to each other?
  - We have seen the Inheritance ("is-a") relationship.
  - We have also seen the Instantiation ("instance-of") relationship.
  - We will now look at a few other relationships that are fundamental to OO design.

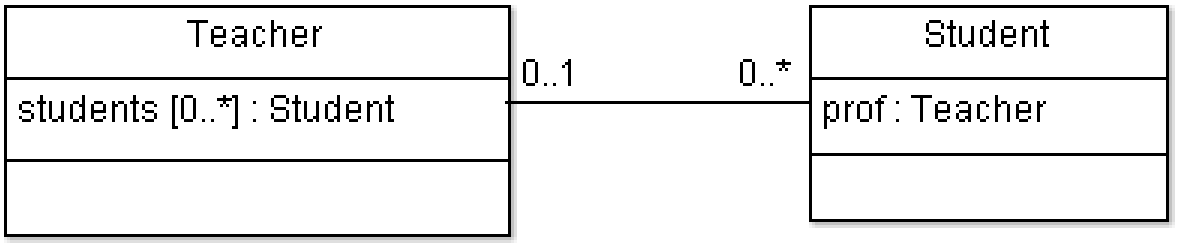
Composition and Aggregation

- These relationships create a complex class from one or more existing classes (or instances), in a whole/part relationship.
- Composition ("owns-a", "has-a"):
  - An object is a component of at most one composition.
  - When a composition instance is destroyed, all objects belonging to this instance are destroyed as well.
  - Example: **`SimCard`** has-a SIM
- Aggregation ("has-a"):
  - An object can be in many aggregations.
  - Example: **`Pond`** has-a **`Duck`**



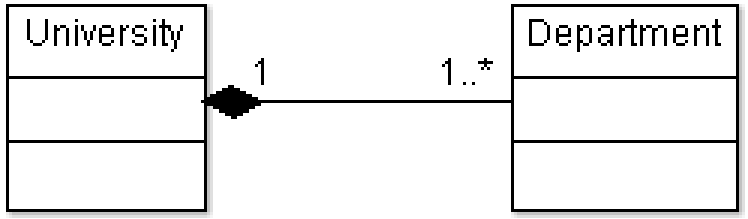
Association

- In an association, an instance of one class is linked to an instance of another class.
  - An aggregation is an association, because the aggregating class has instance variables which refer to objects of its parts.
  - A composition is an association, because the "container" or "owner" has references to its parts,
- An association may have no "container", "owner", or "whole".
  - Example: every teacher has 0 or more students, and every student has 0 or 1 teachers.



Multiplicities

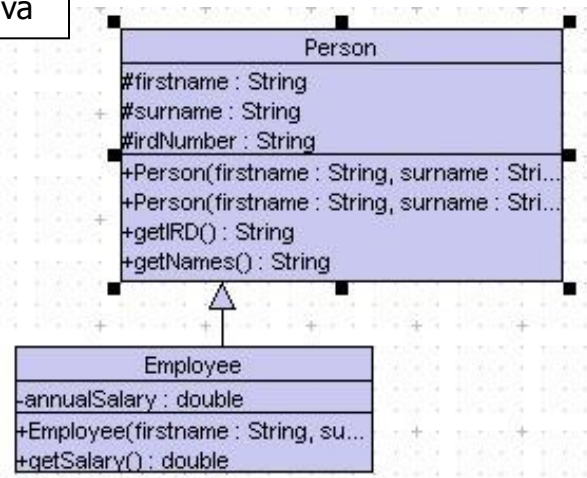
- The multiplicity of an association may be important enough to include in a high-level design document.
  - The filled-diamond notation for compositions implies that the "whole" (Composite) class has a multiplicity of 1..1 or 0..1 – because each Part can belong to at most one whole.



  - No department can exist unless it is part of a university, so the University's multiplicity in this association is 1..1 (sometimes written "1").
  - A university must have at least one department.
- Multiplicities of 0..* are not very informative.

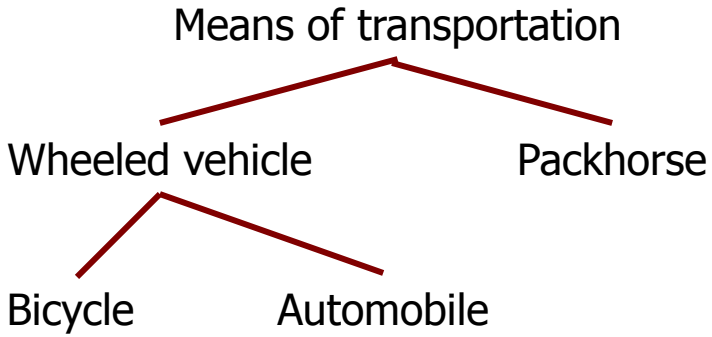
Inheritance

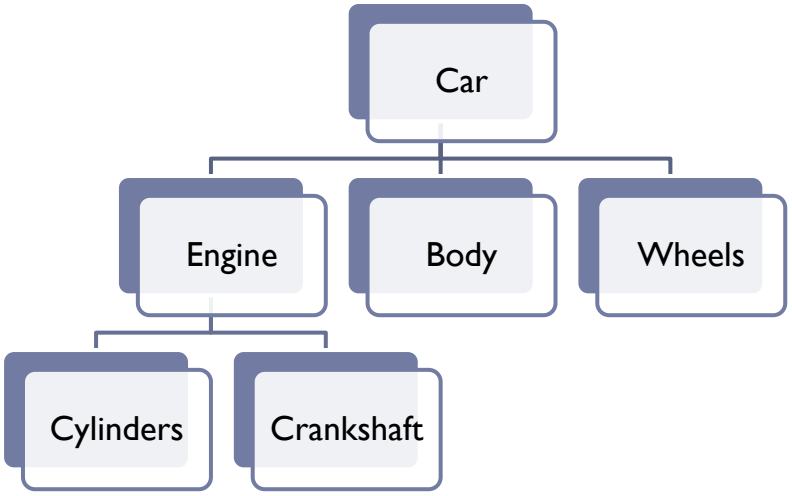
Example: Person.java & Employee.java

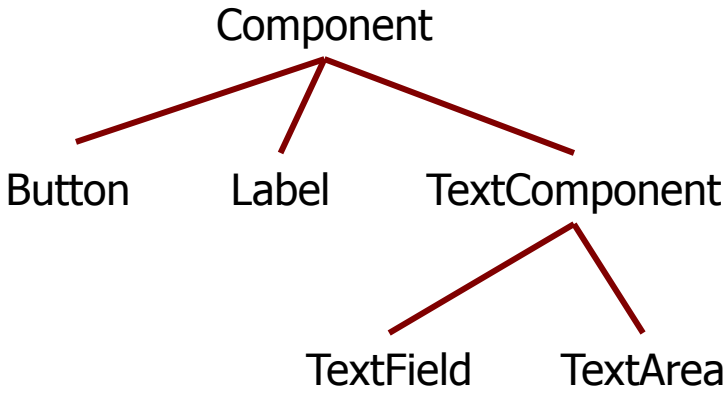


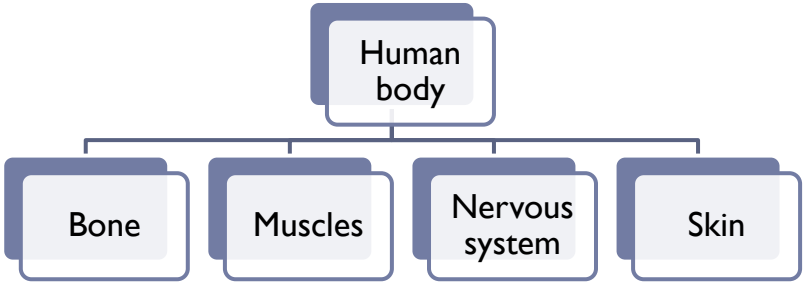
- Generalisation
  - Look for conceptual commonalities in the abstraction
    - Common attributes/state
    - Common methods or behaviour
- "is-a" relationship
  - Subclass "is-a" kind of Superclass
- Superclass (base class) defines the general state and behaviour
- Subclass (derived) class :
  - Created from an existing class, by extending the existing class, rather than rewriting it
- Examples
  - Person
    - Employee & Customer (Customer "is-a" Person)

Composition Vs Inheritance









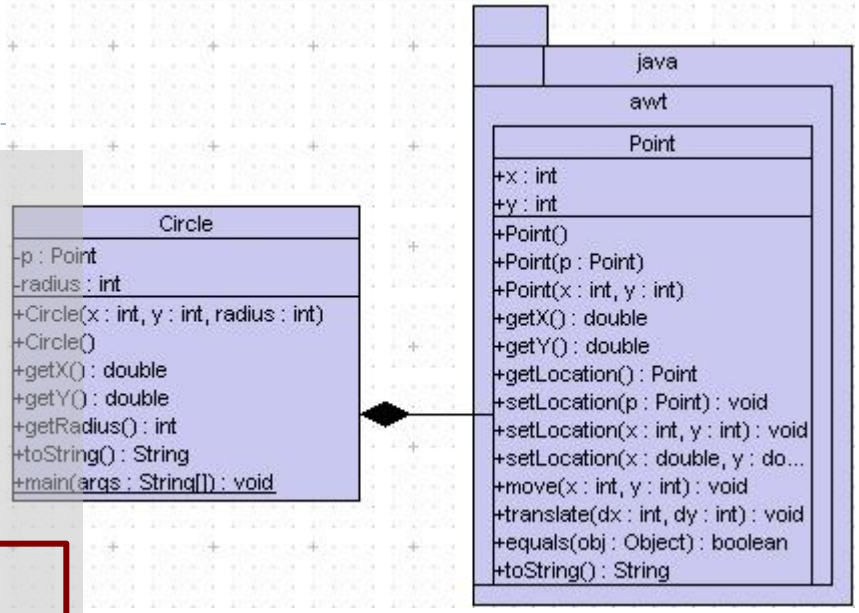
Composition Vs Inheritance

- Suppose we want to create a class to represent circles:
  - Composition or Inheritance?
    - Circle is a point with a radius?
    - Circle has a point with a radius?
- General guideline
  - Inheritance : Is-a
  - Composition : has-a
- But there is no rulebook - the objective is clean, understandable and maintainable code that can be implemented in reasonable time
- Some more guidelines:
  - Inheritance provides a means for constructing highly reusable components, but needs to be used very carefully
  - Choose composition first when creating new classes from existing classes. You should only used inheritance if it is required by your design. If you use inheritance where composition will work, your designs will become needlessly complicated

Composition

Example: Circle_a/Circle.java

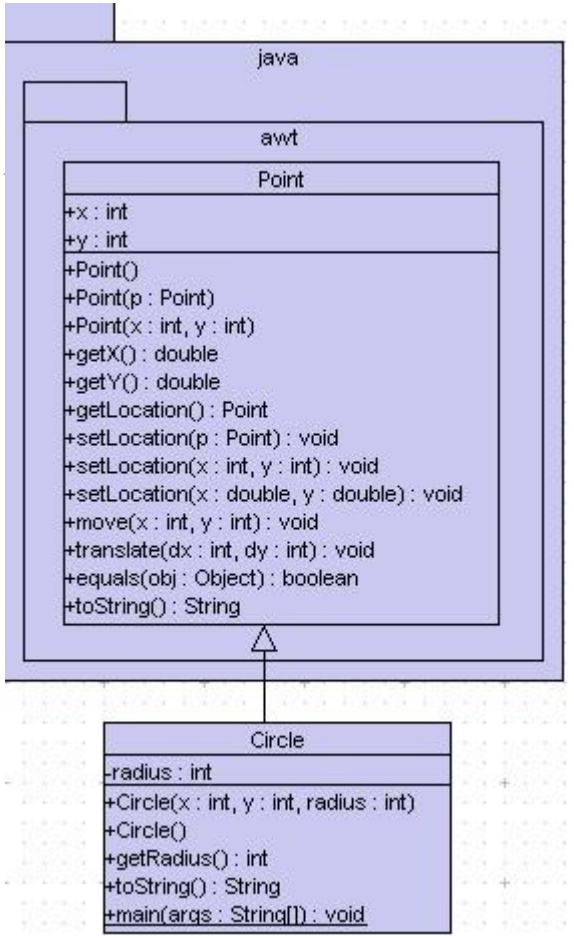
```
import java.awt.Point;
public class Circle {
  private Point p;
  private int radius;
  public Circle (int x, int y, int radius) {
    p = new Point (x, y);
    this.radius = radius;
  }
  public double getX() {
    return p.getX ();
  }
  public double getY() {
    return p.getY ();
  }
  public int getRadius () {
    return radius;
  }
   // additional code
}
```




```
Circle c1 = new Circle();
System.out.println("x=" + c1.getX() + ", y=" + c1.getY());
System.out.println("radius=" + c1.getRadius());
```

Inheritance

Example: Circle_i/Circle.java



- We could reuse Point class since it already has code for representing a point position

```
import java.awt.Point;
public class Circle extends Point {
  private int radius;

  public Circle() {
  }
  public int getRadius () {
    return radius;
  }
  // additional code
}
```

Instance variable: radius

Inherited from Point class

```
Circle c1 = new Circle();
System.out.println("x=" + c1.getX() + ", y=" + c1.getY());
System.out.println("radius=" + c1.getRadius());
```

Review

- Abstraction
  - The ability of a language (and a designer) to take a concept and create an abstract representation of that concept within a program
- Information Hiding
  - How well does this language, designer, and programmer hide an object's internal implementation?
- Inheritance
  - The "is-a" relation: important for code reuse
- Polymorphism
  - How does this language let us treat related objects in a similar fashion?
- Composition, Aggregation, Association
  - Types of "has-a" relations: ways to build complex classes from simpler ones.