Inheritance and Polymorphism

Is-a Vs. Has-a

- Hierarchical classifications are common
  - A poodle is a kind of dog, so anything any dog can do, a poodle can do (but not the other way around).
  - Pizza is a kind of food; anything you can do to any food item, you can do to pizza (but not the other way around).
- Many modern OO programming languages allow classes to be organized in such a way
  - This is called Inheritance
- Inheritance allows programmers to specify an is-a relationship.
- Composition (a class containing an instance variable of an object) represents a has-a relationship

Abstract Methods and classes

- Sometimes, base or parent classes want to declare functionality but not define it
- A method may be `abstract` and have no method body
  - All classes that `extend` the parent class *must* override the abstract method

Abstract Method

```
public abstract double MonthlySalary();
```

- A class may be abstract as well
  - It must be a base class; no instances may be created
  - usually has at least one abstact method

Abstract Method

```
public abstract class Employee{/*...*/}
```

- Classes that are not abstract are called concrete classes

Interfaces

- Classes usually have *state* (instance variables) and *functionality* (member methods)
- Java allows creation of Interfaces
  - Boundary between two entities
  - Defines *how* objects interact with everything
  - In an interface:
    - All methods are abstract
    - All methods are public
    - There are no instance variables
    - You may define *default methods (Java 8 +)*
  - `public` and `abstract` may be omitted when writing interfaces as they are default
  - Constants are `public static final` by default, so they may be omitted as well
  - These will crop up many times this semester

Why Interfaces?

- Classes `implement` interfaces
  - This allows a formal definition of behavior
  - All methods in the interface *must* be defined/implemented
- Useful for creating *frameworks*
  - Definitions of related interfaces and classes
- Gets around the multi-inheritance problem

Multi-inheritance

Consider the following scenario:

- Class `par1` implements a method with the signature `public static void foo()`
- Class `par2` also implements a method with the signature `public static void foo()`
- Class `child` inherits from both `par1` and `par2`, and does not override `public static void foo()`
- An instance of `child` calls the `foo()` method

Given that when a method is not overridden in the child class, it calls the parent's method, which `public static void foo()` method gets called?

Multi-inheritance

- This ambiguity is caused by *multi-inheritance*
  - `child` inherits from both `par1` and `par2`
- Java gets around this ambiguity by *not allowing multiple inheritance*
  - A class may `extend` at most one other `class`
- Java allows the representation of multiple is-a relationships with interfaces
  - A class may `implement` as many `interfaces` as it wants

Implementing Multiple Interfaces

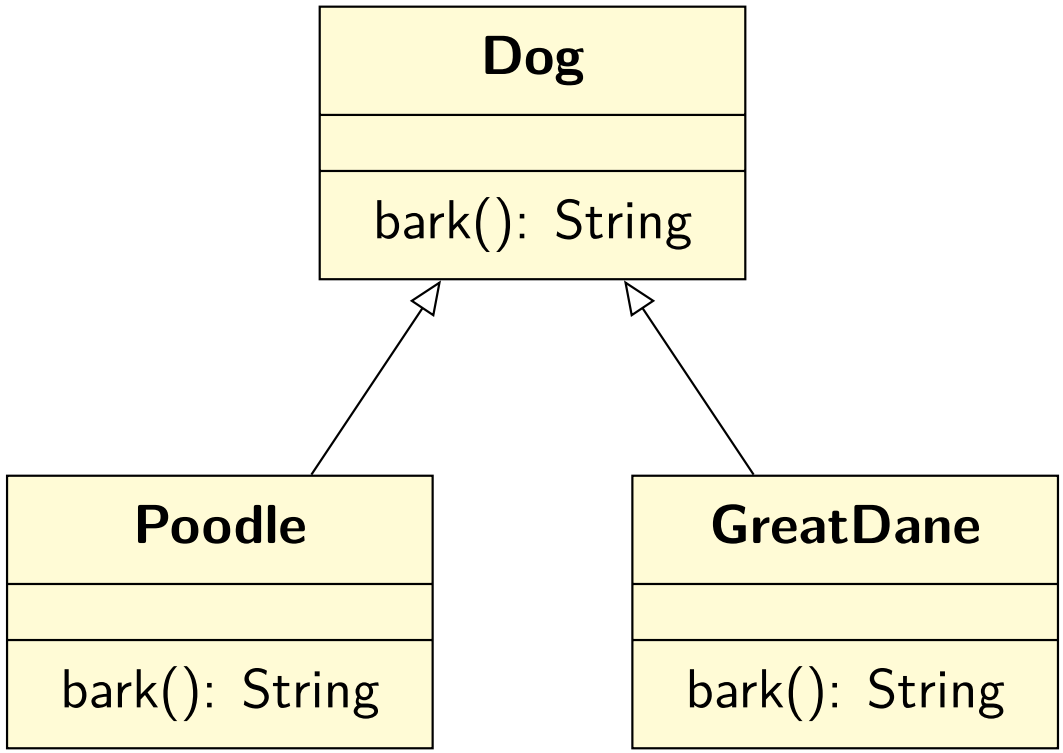
```
public class StudentEntrepreneur extends Student
                    implements Employer, Employee {/*...*/}
```

Inheritance and Interface Rules

- A class may `extend` at most one class
- A class may `implement` any number of interfaces
  - A class must implement all methods (publicly) in the interfaces it implements
- An interface can extend multiple interfaces
  - A class may NOT extend an interfaces
- Instances of interfaces may not exist
  - `new MyInterface()` is not allowed
- Variables that are references to interfaces may exist
  - `MyInterface var = new MyClass()` is allowed, assuming `MyClass implements MyInterface`
- Java determines which method to call based on the type of the object
  - This is the basis of Polymorphism

Polymorphism

Consider the following UML



Polymorphism

```
Dog show[] = new Dog[4];
show[0] = new Poodle();
show[1] = new Poodle();
show[2] = new GreatDane();
show[3] = new GreatDane();

for (Dog d: show)
    System.out.println(d.bark());
```

Java determines which `bark()` method to call based on the type of the instance of the class; to that end it uses the `instanceof` operator

instanceof

```
show[0] instanceof Poodle;
```

Default Methods

Java now allows you to provide a default behavior *in an interface* that runs if the method is not defined in the implementing class

```
public interface MyInterface
{
    default String foo() {/*...*/}
}
```