CSE 143 Java

Inheritance

Reading: Ch. 9, 14

Composition – "has a"

- Classes and objects can be related in several ways
- One way: *composition*, *aggregation*, or *reference*
  - one object's instance variable refers to another object
  - a "has-a" relation
- Simple example: objects representing people

```
public class Person {
    private String name;        // this person's name
    private Person mother;      // this person's mother
}
```


Specialization – "is a"

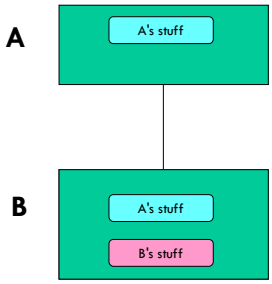
- Two classes &/or interfaces can be related via *specialization*
  - one class/interface is *a special kind of* another class/interface
- Specialization relations can form *classification hierarchies*, just as in the world that the classes are modeling
  - cats and dogs are special kinds of mammals; mammals and birds are special kinds of animals; animals and plants are special kinds of living things
  - lines and triangles are special kinds of polygons; rectangles, ovals, and polygons are special kinds of shapes
- Specialization is not the same as composition
  - A cat "is-an" animal vs. a cat "has-a" tail

Inheritance

- Java (and C++ and many other languages) provide direct support for "is-a" relations
- class ***inheritance***
  - one class can **inherit from** another class, meaning that it's is a special kind of the other
- Specializing class inherits all instance variables and methods of the inherited class
  - Can add additional methods and instance variables
  - Can provide different versions of inherited methods
- Key concept for **object-oriented programming**

Interfaces vs. Class Inheritance

- An interface is a simple form of inheritance
- If B implements interface A, then B inherits the stuff in A (which is nothing but the method signatures of B)
- If B extends class A, then B inherits the stuff in A (which can include method code and instance variables)



Example: Representing Animals

- Generic Animal

```
public class Animal {
   private int numLegs;

   /** Return the number of legs */
   public int getNumLegs( ) {
      return this.numLegs;
   }

   /** Return the noise this animal makes */
   public String noise( ) {
      return "?";
   }
}
```

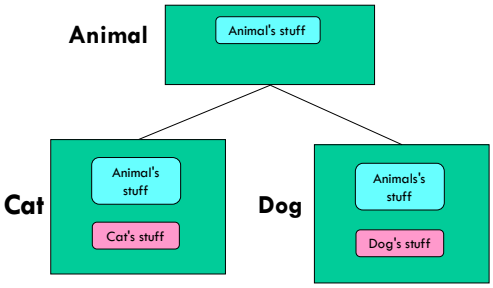
Specific Animals

- Cats

```
public class Cat extends Animal {
   // inherit numLegs and getNumLegs()

   // additional inst. vars and methods
   ....

   /** Return the noise a cat makes */
   public String noise( ) {
      return "meow";
   }
}
```

- Dogs

```
public class Dog extends Animal {
   // inherit numLegs and getNumLegs()

   // additional inst. vars and methods
   ....

   /** Return the noise a dog makes */
   public String noise( ) {
      return "WOOF!!";
   }
}
```



Vocabulary and Principles

- If class D extends/inherits from B
  - B is called the ***superclass*** (sometimes called the base class)
  - D is called the ***subclass*** (or derived class)
- Class D ***inherits*** all methods and fields from class B
  - But not constructors or static methods or static fields
- Class D may contain additional (new) methods and fields
  - But may not delete any
- Key fact: every object of type D is also an object of type B
  - D can do anything that B can do (because of inheritance)
  - D can be used in any context where a B object is appropriate

Method Overriding

- If class D extends B, class D may provide an *alternative*, *replacement* implementation of any method it would otherwise inherit from B
  - The definition in D is said to ***override*** the definition in B
- An overriding method cannot change the number of arguments or their types, or the type of the result [why?]
  - can only provide a different body
- Cannot override an instance variable

Polymorphism

- *Polymorphic*: "having many forms"
- A variable that can refer to objects of different types is said to be *polymorphic*
- Methods with polymorphic arguments are also said to be polymorphic

```
public void speak(Animal a) {
    System.out.println(a.noise( ));
}
```

- Polymorphic methods can be *reused* for many types

Static and Dynamic Types

- With polymorphism, we can distinguish between
  - Static type: the declared type of the variable (fixed during execution)
  - Dynamic type: the run-time class of the object the variable currently refers to (can change as program executes)

```
public void speak(Animal a) {
    System.out.println(a.noise( ));
}

Cat foofoo = new Cat( );
speak(foofoo);

Dog fido = new Dog( );
speak(fido);
```


Method Lookup & Dynamic Dispatch

- When a message is sent to an object, the right method to invoke is the one in the *most specific class* that the object is an instance of
  - Makes sure that method overriding always has an effect
- Method lookup (a.k.a. **dynamic dispatch**) algorithm:
  - Start with the *run-time class* of the receiver object (not the static type!)
  - Search that class for a matching method
  - If one is found, invoke it
  - Otherwise, go to the superclass, and continue searching
- Example:

```
Animal a = new Cat( );
System.out.println(a.noise( ));
a = new Dog( );
System.out.println(a.getNumLegs( ));
```


Summary

- Object-oriented programming is huge
  - Lots of new concepts and terms
  - Lots of new programming and modeling power
- Ideas (so far!)
  - Composition ("has a") vs. specialization ("is a")
  - Inheritance
  - Method overriding
  - Polymorphism, static vs. dynamic types
  - Method lookup, dynamic dispatch