CSC 143 Java
Inheritance
Reading: Ch. 10

Composition: "has a"
- Classes and objects can be related in several ways
- One way: composition, aggregation, or reference
- Dog has-a owner, dog has legs, dog has collar, etc.
- In java: one object refers to another object
  - via an instance variable

```java
public class Dog {
    private String name; // this dog's name
    private int age; // this dog's age
    private Person owner; // this dog's owner
    private Dog mother, father; // this dog's parents
    private Color coatColor; // etc, etc.
}
```

- One can think of the dog as "composed" of various objects: "composition"

Picturing the Relationships
- Dog Fido; // might be 6 years old, brown, owned by Marge, etc.
- Dog Apollo; // might be 2 years old, missing a leg, etc.
- In Java, it is a mistake to think of the parts of an object as being "inside" the whole.

Drawing Names and Objects
- Names and objects
  - Very different things!
  - In general, names are applied to objects
  - Objects can refer to other objects using instance variable names

```
Fido (a name) refers to another object of type Dog
Dog age 6 mother

MyDoggie null
Fido refers to
```

Specialization – "is a"
- Specialization relations can form classification hierarchies
  - 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
- Keep in mind: Specialization is not the same as composition
  - A cat "is-an" animal vs. a cat "has-a" tail
"is-a" in Programming

- Classes &/or interfaces can be related via specialization
  - one class/interface is a special kind of another class/interface
  - Rectangle class is a kind of Shape
- So far, we have seen one Java technique to capture this idea: interfaces
  - Java interfaces are one special case of a more general design approach: Inheritance

Inheritance

- Java provides direct support for "is-a" relations
  - likewise C++, C#, and other object-oriented languages
- Class inheritance
  - one class can inherit from another class, meaning that it's a special kind of the other
- Terminology
  - Original class is called the base class or superclass
  - Specializing class is called the derived class or subclass

Inheritance: The Main Programming Facts

- Subclass inherits all instance variables and methods of the inherited class
  - All instance variables and methods of the superclass are automatically part of the subclass
  - Constructors are a special case (later)
- Subclass can add additional methods and instance variables
- Subclass can provide different versions of inherited methods

B extends A

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)
- To distinguish the two, people sometimes say "interface inheritance" vs. "class inheritance"
- What if you heard the phrase "code inheritance"?

Example: Representing Animals

- Generic Animal
  ```java
  public class Animal {
    private int numLegs;
    /** Return the number of legs */
    public int getNumLegs() { return this.numLegs; }
  }
  ```
  ```java
  /** Return the noise this animal makes */
  public String noise() { return "?"; }
  ```
Specific Animals

- **Cats**
  ```java
  public class Cat extends Animal {
    // inherit numLegs and getNumLegs()
    // additional inst. vars and methods
    public String noise() {
      return "meow";
    }
  }
  ```

- **Dogs**
  ```java
  public class Dog extends Animal {
    // inherit numLegs and getNumLegs()
    // additional inst. vars and methods
    public String noise() {
      return "WOOF!!";
    }
  }
  ```

More Java

If class D extends B/inherits from B...
- Class D **inherits** all methods and fields from class B
- But... "all" is too strong
  - constructors are not inherited
  - same is true of static methods and static fields
    although these static members are still available in the subclass
- Class D may contain additional (new) methods and fields
- But has no way to delete any methods or any fields of the B class
  (though D can override methods from B (very common) and hide
  fields (not recommended))

Never to be Forgotten

If class D extends/inherits from B...
- a D can do anything that a B can do (because of inheritance)
- a D can be used in any context where a B 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
- Can you override an instance variable?
  - Not exactly...

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
  ```java
  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)

```java
public String noise() {
    // this has static type Animal
    ...
}

Cat foofoo = new Cat();
foofoo.noise();  // inside noise(), this has dynamic type Cat

Dog fido = new Dog();
fido.noise();  // inside noise(), this has dynamic type Dog
```

Dynamic Dispatch

- "Dispatch" refers to the act of actually placing a method in execution at run time
- When types are static, the compiler knows exactly what method must execute.
- When types are dynamic... the compiler knows the name of the method – but there could be ambiguity about which version of the method will actually be needed at run time.
  - In this case, the decision is deferred until run-time, and we refer to it as dynamic dispatch

Method Lookup: How Dynamic Dispatch Works

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

Summary

- Object-oriented programming is hugely important
  - Lots of new concepts and terms
  - Lots of new programming and modeling power
  - Used more and more widely

- Ideas (so far)
  - Composition ("has a") vs. specialization ("is a")
  - Inheritance
  - Method overriding
  - Polymorphism, static vs. dynamic types
  - Method lookup, dynamic dispatch