Week 8 Description

This week, we see that you do not always have to start from scratch when creating a new user-defined class. Often, we use an existing class as a starting point, and modify it to create a new class. This process is called Inheritance, and it's the another one of the big deals in Object Oriented programming. We'll discuss enumerated types.

Inheritance

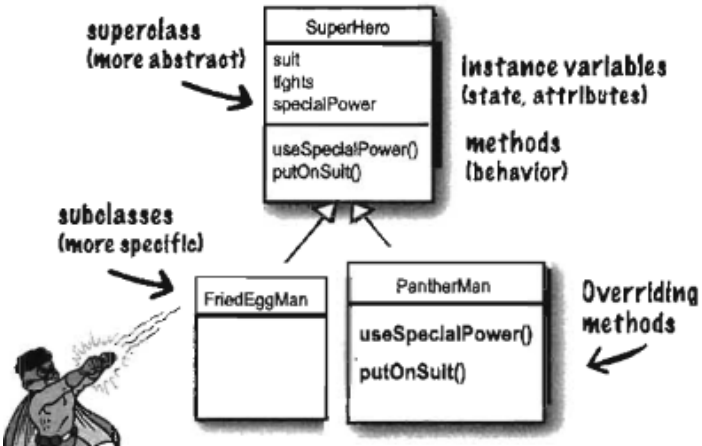
Inheritance *allows an object of one class to acquire the properties and methods of another class*.

Understanding Inheritance

When you design with inheritance, you put common code in a class and then tell other more specific classes that the common (more abstract) class is their superclass. When one class inherits from another, **the subclass inherits from the superclass.**

In Java, we say that the **subclass *extends* the superclass**. An inheritance relationship means that the subclass inherits the **members** of the superclass. When we say "members of a class" we mean the instance variables and methods.

For example, if PantherMan is a subclass of SuperHero, the PantherMan class automatically inherits the instance variables and methods common to all superheroes including `suit`, `tights`, `specialPower`, `useSpecialPower()` and so on. But the PantherMan **subclass can add new methods and instance variables** of its own, and it **can override the methods it inherits from the superclass** SuperHero.



FriedEggMan doesn't need any behavior that's unique, so he doesn't override any methods. The methods and instance variables in SuperHero are sufficient.

PantherMan, though, has specific requirements for his suit and special powers, so `useSpecialPower()` and `putOnSuit()` are both overridden in the PantherMan class.

Instance variables are not overridden because they don't need to be. They don't define any special behavior, so a subclass can give an inherited instance variable any value it chooses. PantherMan can set his inherited `tights` to purple, while FriedEggMan sets his to white.

Inheritance is central to object-oriented (OO) programming in Java and other OO languages. The key concept is creating a new class from an existing one. The new class is called a **subclass** and the original one a **superclass**.

Here's another example:

:oordinates for where the animal is

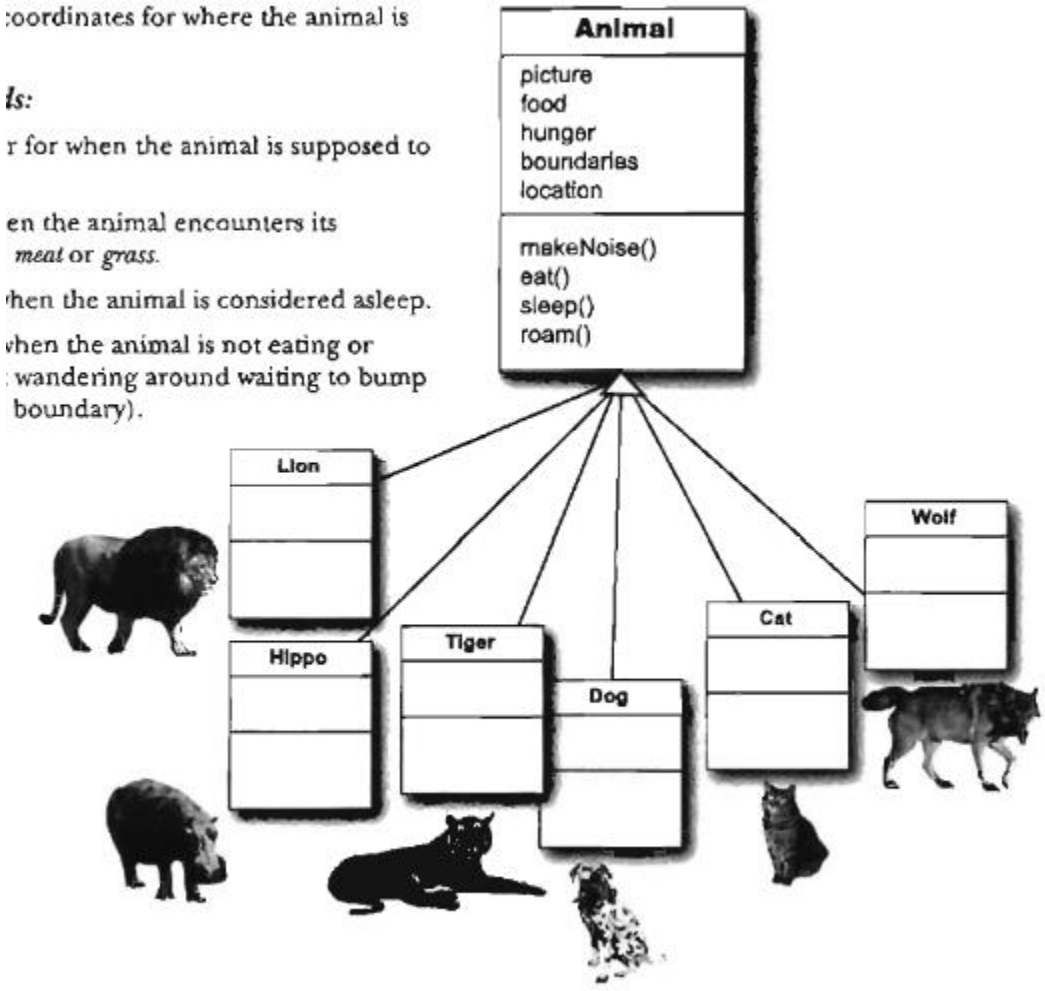
ls:

r for when the animal is supposed to

en the animal encounters its
meat or *grass*.

hen the animal is considered asleep.

vhen the animal is not eating or
wandering around waiting to bump
boundary).



If class B extends class A, class B IS-A class A.

This is true anywhere in the inheritance tree. If class C extends class B, class C passes the IS-A test for both B *and* A.

Canine extends Animal

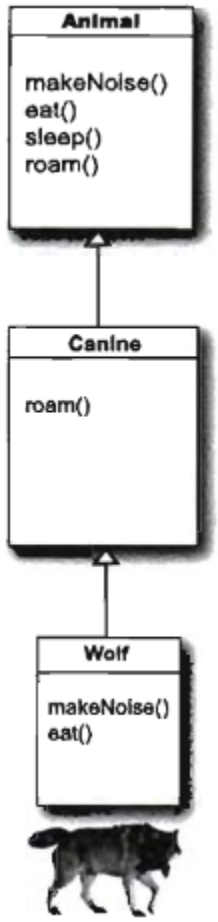
Wolf extends Canine

Wolf extends Animal

Canine IS-A Animal

Wolf IS-A Canine

Wolf IS-A Animal



With an inheritance tree like the one shown here, you're *always* allowed to say **"Wolf extends Animal"** or **"Wolf IS-A Animal"**. It makes no difference if Animal is the superclass of the superclass of Wolf. In fact, **as long as Animal is *somewhere* in the inheritance hierarchy above Wolf, Wolf IS-A Animal will always be true.**

The structure of the Animal inheritance tree says to the world:

"Wolf IS-A Canine, so Wolf can do anything a Canine can do. And Wolf IS-A Animal, so Wolf can do anything an Animal can do."

It makes no difference if Wolf overrides some of the methods in Animal or Canine. As far as the world (of other code) is concerned, a Wolf can do those four methods. *How* he does them, or *in which class they're overridden* makes no difference. A Wolf can makeNoise(), eat(), sleep(), and roam() because a Wolf extends from class Animal.

The Object Class

Every class in Java is, either directly or indirectly, a subclass of a Java library class called Object. Therefore, every class inherits the methods of Object. The subclass can either keep these methods unchanged, or override them. Some methods of the Object class:

```
boolean equals (Object obj)
    Returns true if this object is an alias of the specified object.

String toString ()
    Returns a string representation of this object.

Object clone ()
    Creates and returns a copy of this object.
```

Enumerated Types

https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

In few cases, a variable should only hold a restricted set of values. For example, you may want to represent clothes in four sizes: small, medium, large, and extra large. It could be an error-prone setup to encode these sizes as integers 1, 2, 3, 4 or characters S, M, L, XL because there is a possibility for a variable to hold a wrong value such as 0 or m.

In this situation, we can define our own enumerated type. An enumerated type has a finite number of named values or constants. To represent cloth sizes, we can write:

```
enum Size {SMALL, MEDIUM, LARGE, EXTRA_LARGE};
```

To declare a variable of enumerated type, we can write:

```
Size clothSize = Size.MEDIUM;
```

A variable of type Size can hold only one of the values listed in the type declaration, or the special value null that indicates that the variable is not set to any value at all.

Enumeration Classes

We can add constructors, methods, and fields to an enumerated type. Keep in mind that the constructors are only invoked when the enumerated constants are constructed.

```
public enum Size {

    SMALL("S"),MEDIUM("M"), LARGE("L"),EXTRA_LARGE("XL");

    private String abbreviation;

    //constructor of an enumeration is automatically private

    Size(String abbr){

        this.abbreviation = abbr;

    }

    public String getAbbreviation(){

        return abbreviation;

    }

}
```

The constructor of an enumeration is always private. We can omit the private modifier. However, it is a syntax error to declare an enum constructor as public.

All enumerated types are subclasses of the class Enum. Among all the inherited methods, the most useful one is toString. Here, toString() returns the name of the enumerated constant. For instance, Size.SMALL.toString() returns the string “SMALL”.