Static, Dynamic, and Inclusion Polymorphism

Lesson 4.1

Learning Outcomes

LO 4.1.1 **Identify** the correct type of polymorphism applied on object identifiers

LO 4.1.2 **Implement** valid programming statements/ functions that apply static, dynamic, and/or inclusion polymorphism

Polymorphism

Polymorphism is derived from the Greek word *poly* – meaning many, and *morphe* – meaning form.

In OOP, there are 4 major kinds of polymorphism:

1. **Static polymorphism** *(classes)*
2. **Dynamic polymorphism** *(inheritance)*
3. **Inclusion polymorphism** *(inheritance)*
4. Parametric polymorphism* *(generics)*

**Will not be discussed in this lesson. Either will be discussed as additional topic of this course or in the future when you tackle advanced object-oriented design.*

Static polymorphism

Static polymorphism is a polymorphism evident during *early binding*.

Early binding means *during compile time*. In Java, static polymorphism is also called *method overloading*.

Method overloading is implementing multiple methods of the same class that uses the *same name* but with *different sets of parameters*.

Static polymorphism

The parameter sets have to differ in at least one of the following criteria:

1. *Number of parameters*
2. *Data types of parameters*
3. *Order of parameters*

Since the sets of parameters for each method overload is different, each method has a *different signature*.

Static polymorphism

Example:

```
public class Add
{
    public int perform(int a, int b);
    public double perform(int a, double b);
    public double perform(double a, double b);
}
```

Dynamic polymorphism

Dynamic polymorphism is a polymorphism evident during *late binding*.

Late binding means this form of polymorphism doesn't allow the compiler to determine the executed method. The method executed is only chosen during *runtime*.

In Java, dynamic polymorphism is represented in the operation of *method overriding* which is a benefit of inheritance.

Inclusion polymorphism

Inclusion polymorphism is another polymorphism evident during *late binding*. Inclusion polymorphism is also known as *subtyping*.

Subtyping, another benefit of inheritance, is the ability of any *ancestor class identifier* to reference a *descendant class object*.

Inclusion polymorphism

Example:

```
public class Animal { … }
public class Dog extends Animal { … }
public class Cat extends Animal { … }
public class Fish extends Animal { … }

Animal a = new Dog();
Animal b = new Cat();

Animal[] animals = new Animal[4];
animals[0] = new Animal();
animals[1] = new Dog();
animals[2] = new Cat();
animals[3] = new Fish();
```

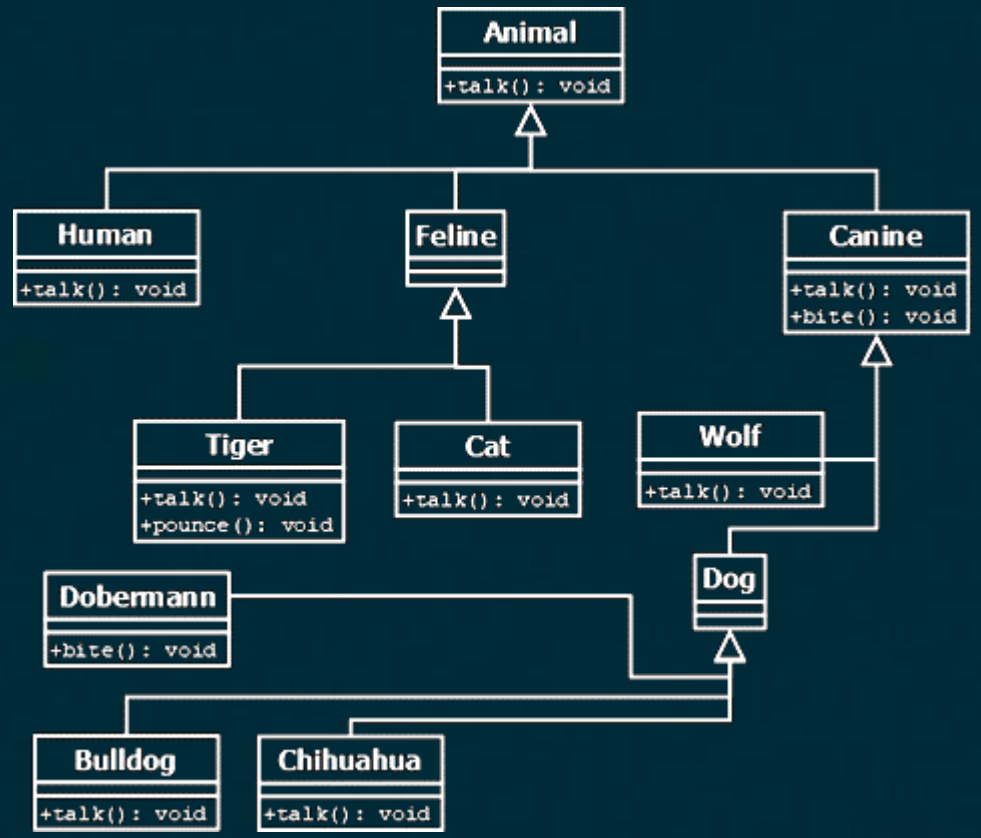
Inclusion polymorphism

If a subtype object is represented by an *ancestor type identifier*, you can only call *methods existing on the ancestor type identifier*.

If a method of that ancestor type is *overridden* along the subclasses leading to the subtype, then the *nearest* class of the subtype along that ancestry that overrides that method is the one that will be executed when that specific method is called.

Inclusion polymorphism

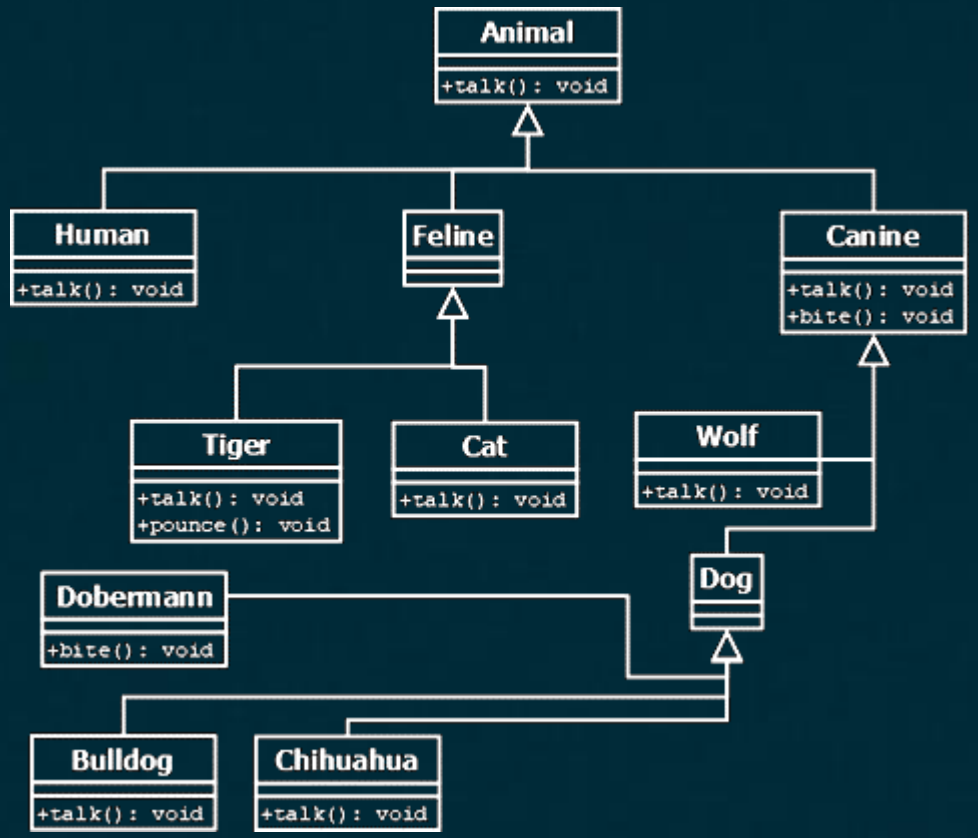
Example:




```
Animal a = new Animal();
a.talk(); //will execute the implementation
          //talk of class Animal

a = new Dog();
a.talk(); //will execute the implementation
          //talk of class Canine

a = new Tiger();
a.talk(); //will execute the implementation
          //talk of class Tiger

a = new Dobermann();
a.talk(); //will execute the
          //implementation talk of
          //class Canine
```

Inclusion polymorphism

Example:




```
Animal a = new Tiger();
a.pounce(); //cause an error during compile
            //time because pounce method does
            //not exist in class Animal

Canine c = new Dobermann();
c.bite();   //will execute successfully the
            //bite method of class Dobermann
            //since the bite method of
            //the same signature exists
            //in class Canine
```

Inclusion polymorphism

To know whether an object is inclusion polymorphic, you can perform an **instanceof** test.

Example:

```
Animal[] animals = new Animal[4];
animals[0] = new Animal();
animals[1] = new Dog();
animals[2] = new Cat();
animals[3] = new Fish();

//x will have the value true because even though animals is
//declared as an array of Animal objects, yet the object inside
//animals[1] is instantiated as a Dog object in line 3 above.
boolean x = animals[1] instanceof Dog;
```

LO 4.1.1 Identify the correct type of polymorphism applied on object identifiers

Identify the parts of the code below applied with polymorphism and of what type assuming that the code was compiled and ran successfully.

```
ArrayList<MenuItem> orders = new ArrayList<MenuItem>();
orders.add(new Appetizer("Caesar Salad", 110.25));
orders.add(new Steak("Pan-seared Filet Mignon", 1210.00, Doneness.MediumRare));
orders.add(new Dessert("Crème Brûlée", 152.75));

for(MenuItem m : orders)
{
    m.prepare();
    if(m instanceof GrilledDish)
        ((GrilledDish)m).grillTo(Grillers.getAvailable());
}
```

LO 4.1.1 Identify the correct type of polymorphism applied on object identifiers

Identify the parts of the code below applied with polymorphism and of what type assuming that the code was compiled and ran successfully.

```
Chef c = new Chef(“Gordon Ramsay”);
Ingredient[] igds = new Ingredient[] {
                        Ingredients.Garlic, Ingredients.Onion, Ingredients.Pork,
                        Ingredients.Fish, Ingredients.Spinach, Ingredients.Egg};

Dish[] o = new Dish[3];
o[0] = c.generateRecipeWithIngredients(igds[0], true);
o[1] = c.generateRecipeWithIngredients(igds[0], igds[3],
                                       Condiment.Salt);
o[2] = c.generateRecipeWithIngredients(igds, CookingType.StirFry);

for(Dish d : o)
    System.out.println(d);
```

LO 4.1.2 Implement valid programming statements/ functions that apply static, dynamic, and/or inclusion polymorphism

Continue implementing the *Plants Vs Zombies* classes from the previous module that necessarily applies at least once: static, dynamic, and inclusion polymorphism.