Fundamentals of Obejct Oriented Programming

LUNCH AND LEARN – WEDNESDAY, MARCH 1, 2017

Basic OOP: Abstraction and Encapsulation

What is Object Oriented Programming?

OOP is the practice of developing code in a manner where the focus is on the data, and the methods that it can perform.

Each object only cares about itself and how it interacts with itself, and other objects.

Objects are generally noun-like; for example, *Pidgeon* or *Dog*.

Even non-physical things and concepts can be represented in an OOP manner:

- Lists are a representation of any ordered set of *things*
- Sets are a representation of any unique set of *things*
- Tweets are a representation of not-funny quips people say and images they post of their food.

What are the fundamentals of OOP?

Abstraction: grouping and representing meaningful data together

Encapsulation: hiding internal logic that the outside world does not care about

Inheritance: objects can be based off other objects

Polymorphism: we can have many ways of doing the same method on a class.

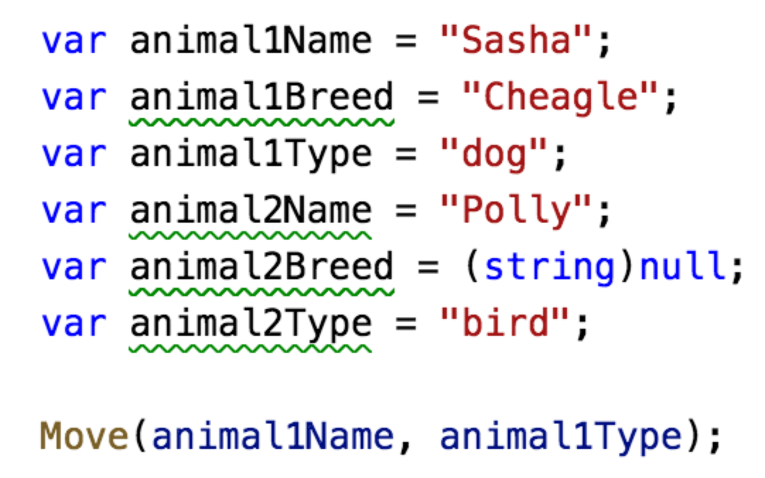
What is abstraction?

Abstraction is the concept of taking your procedures and grouping them together in a reasonable fashion.

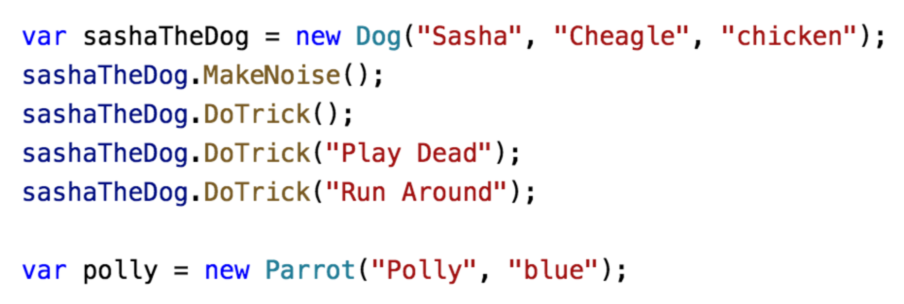
On top, we see a procedural way of writing code without classes or objects. We just see a line by line description of what our program is supposed to do – it is very limited to the current intent of the program, and not very expandable.

Below, we see a grouping based on commonality. We make animals, of different types, by constructing new classes.

Let's take a look at IAnimal.cs in order to see how we'd define an interface for our animal.



```
var animal1Name = "Sasha";
var animal1Breed = "Cheagle";
var animal1Type = "dog";
var animal2Name = "Polly";
var animal2Breed = (string)null;
var animal2Type = "bird";

Move(animal1Name, animal1Type);
```



```
var sashaTheDog = new Dog("Sasha", "Cheagle", "chicken");
sashaTheDog.MakeNoise();
sashaTheDog.DoTrick();
sashaTheDog.DoTrick("Play Dead");
sashaTheDog.DoTrick("Run Around");

var polly = new Parrot("Polly", "blue");
```

Classes and Objects in C#

In C#, we can practice Object Oriented programming by writing classes and making objects.

Classes are a blueprint for a complex piece of data. Classes themselves are types just like integers and booleans, and don't actually have any data of their own (with the exception of static data). Instead, they have properties and methods which describe data that instances of your objects will contain and functions they will perform.

If you look in your *classes* folder, you'll see all our classes for different types of animals!

While classes are blueprints, we make new instances of the classes (see Program.cs). You can tell you are making an instance of a class by the use of the *new* keyword, followed by the constructor function of a class.

These instances are self contained objects that have all the data and methods of that class. You can get or set the properties of that class instance and not affect other instances of that class.

What is encapsulation?

Encapsulation is the process of hiding internal data and logic from the outside. It can be seen as part of DOING the abstraction

Real world example: you don't need to know how a dog's body works to know that when you feed your dog a steak, they eat the steak and digest it. (See dog.js *Eat* method for an abstraction of that!)

Real world example: You also don't know how the parrot will decide to speak, but it will emulate something spoken to. You just want to get the result of it speaking.

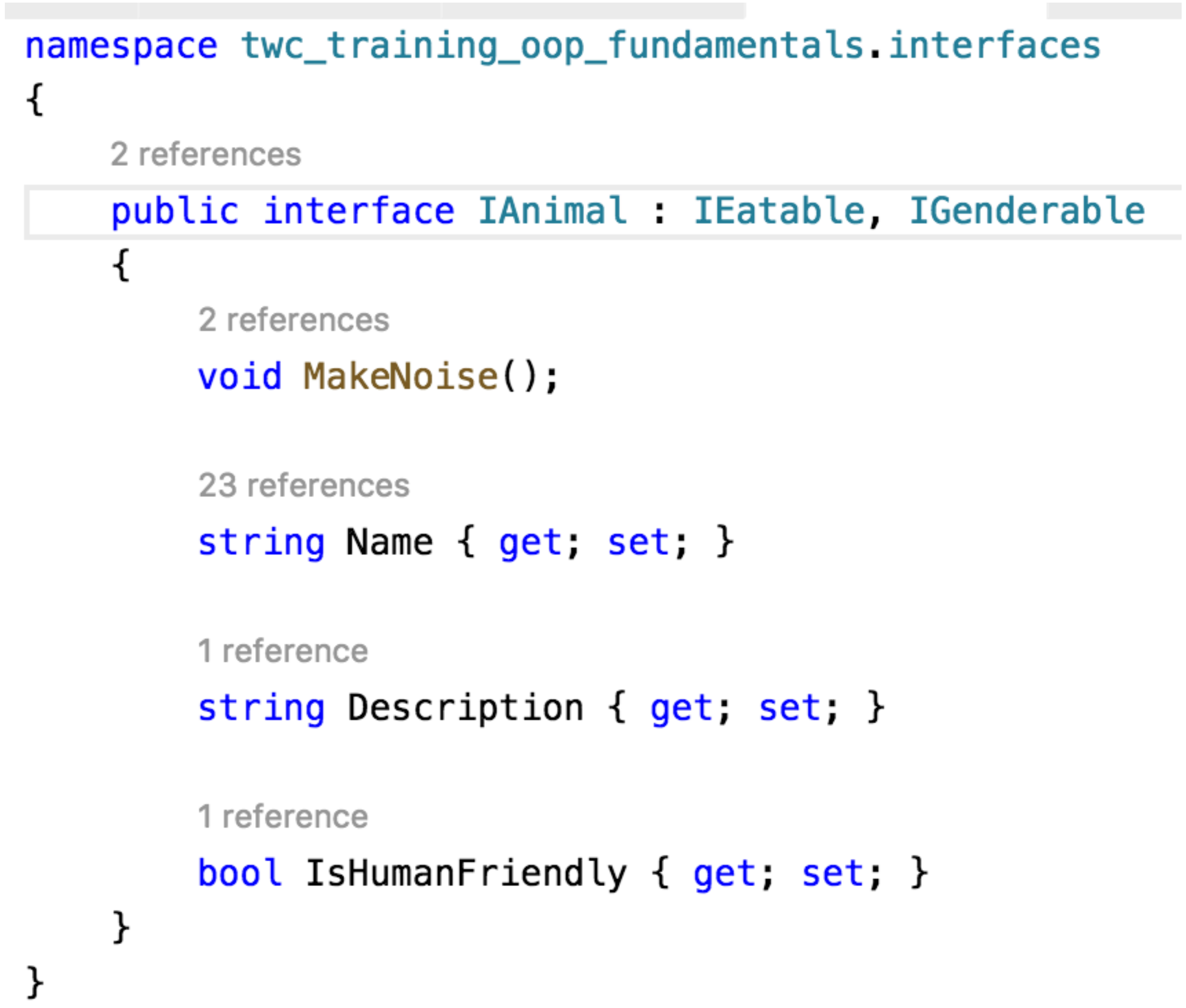
This is useful because it allows us to focus on making objects that are self-contained: they perform what is expected of them, and we don't need to know how they do that.

What is an interface?

An interface is a contract that describes what a class has and how it is expected to behave.

When we make animals, they would implement IAnimal, which means they would have all the properties and methods defined in Ianimal.

Interfaces can implement other interfaces, as well!



```
namespace twc_training_oop_fundamentals.interfaces
{
    2 references
    public interface IAnimal : IEatable, IGenderable
    {
        2 references
        void MakeNoise();

        23 references
        string Name { get; set; }

        1 reference
        string Description { get; set; }

        1 reference
        bool IsHumanFriendly { get; set; }
    }
}
```

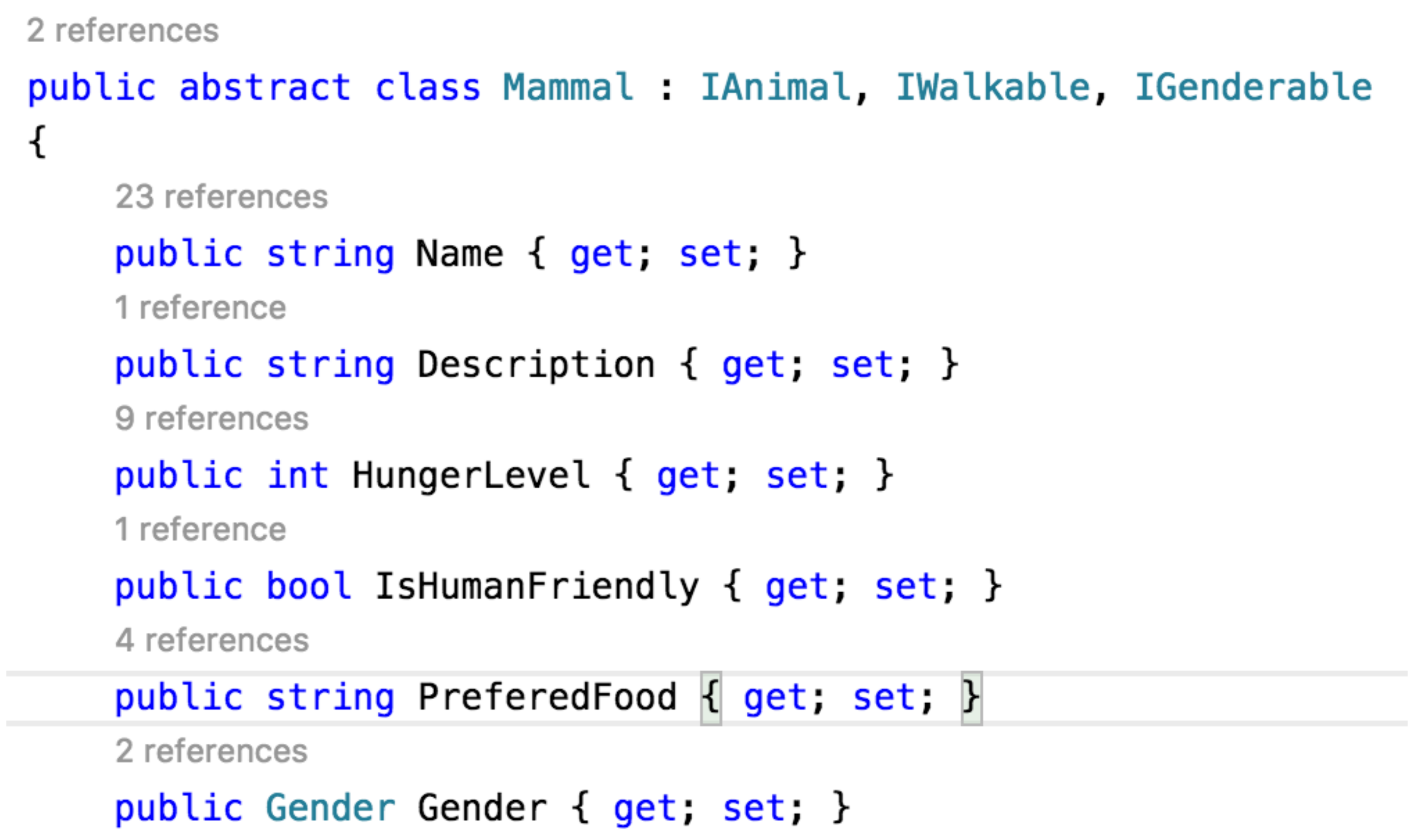
How do we implement an interface in C#?

Implementation is a fulfillment of an interface; to implement an interface, your class needs to have all the properties and methods defined in the interface.

Our *Mammal* class is going to implement 3 interfaces:

- IAnimal
- IGenderable
- IWalkable

As such, it must implement all methods and all properties from all those interfaces.



```
2 references
public abstract class Mammal : IAnimal, IWalkable, IGenderable
{
    23 references
    public string Name { get; set; }
    1 reference
    public string Description { get; set; }
    9 references
    public int HungerLevel { get; set; }
    1 reference
    public bool IsHumanFriendly { get; set; }
    4 references
    public string PreferedFood { get; set; }
    2 references
    public Gender Gender { get; set; }
```

Constructors and initialization

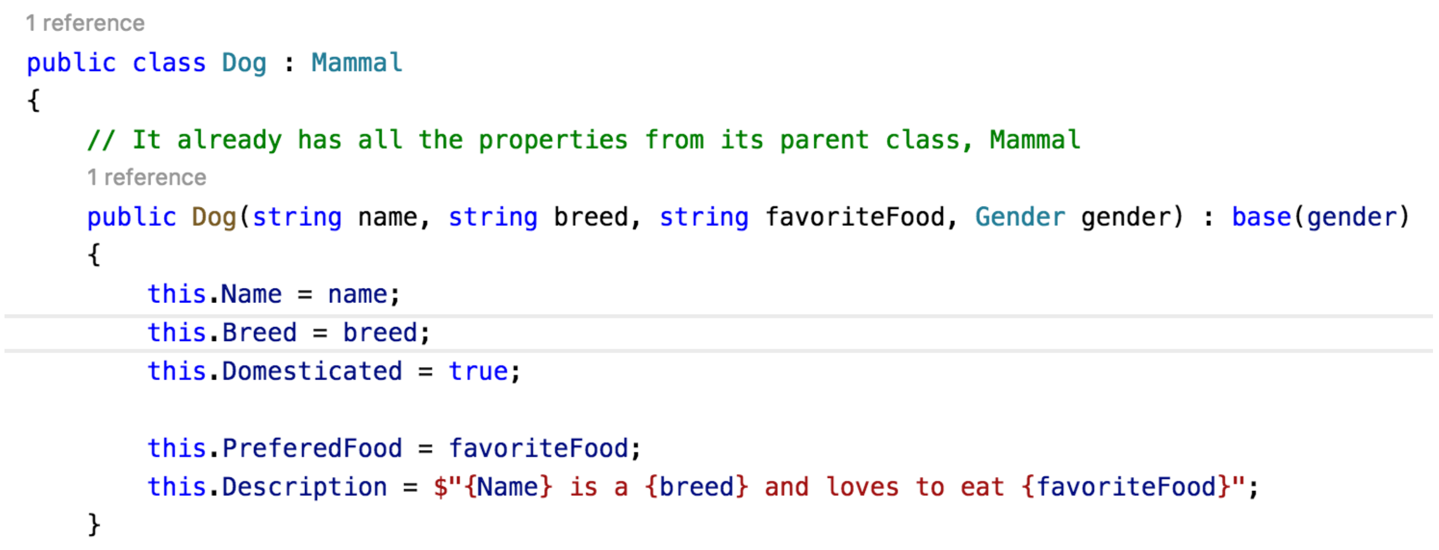
When creating a class, we create special methods call *constructors* that are called when we make a new **instance** of the class. The constructor method name is the same as the class name.

Constructors are extremely important, as they allow you to setup all the data for **each instance of the class**.

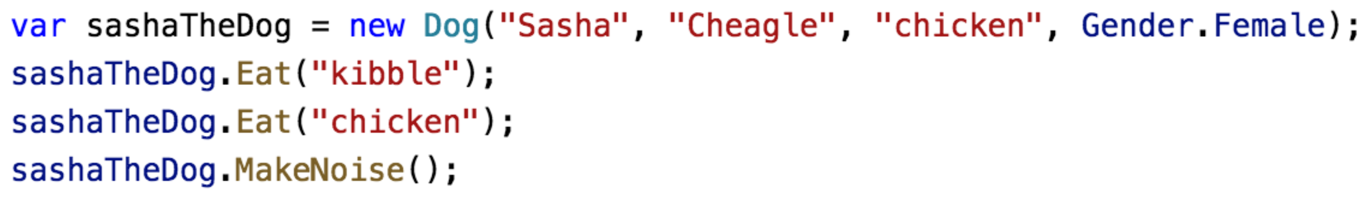
We often pass data about the instance of that class into the constructor, so that we can setup that instance with relevant data.

Here, we see that the Dog has a constructor method that takes the dog's name, breed, favorite food, and gender and sets up the properties of that class instance based on those parameters.

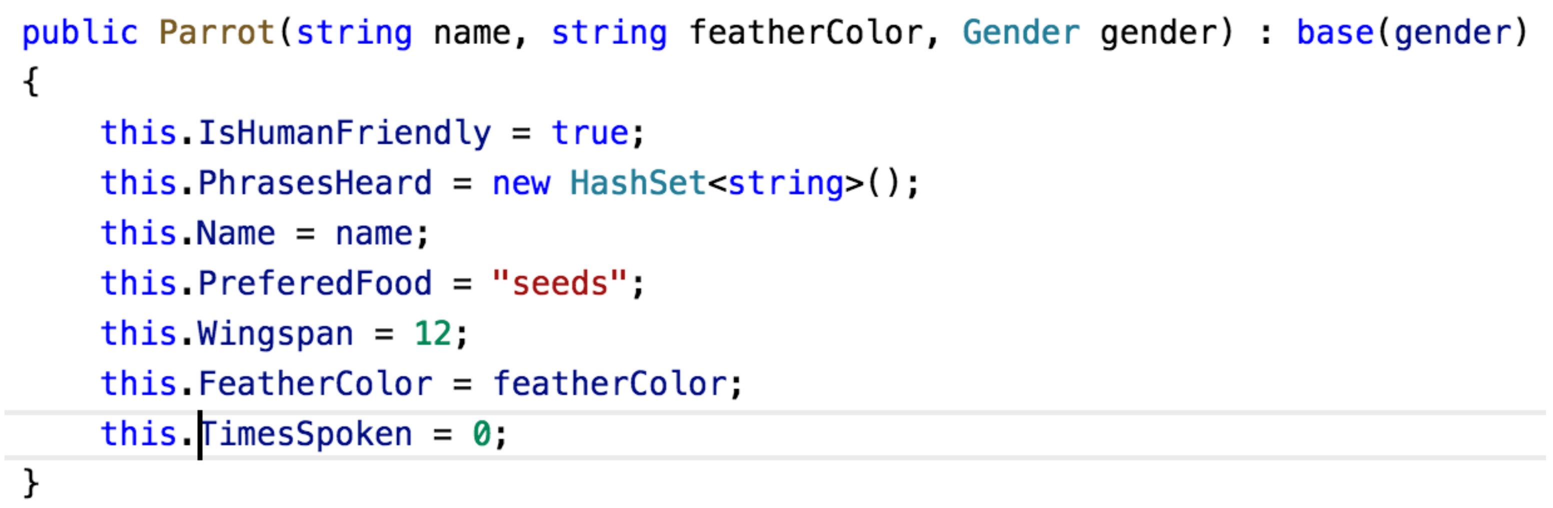
Constructors also allow us to initialize data that does not come from the parameters, such as setting the parrot's PhrasesHeader property to a brand new hash set at the time of construction.



```
1 reference
public class Dog : Mammal
{
    // It already has all the properties from its parent class, Mammal
    1 reference
    public Dog(string name, string breed, string favoriteFood, Gender gender) : base(gender)
    {
        this.Name = name;
        this.Breed = breed;
        this.Domesticated = true;

        this.PreferedFood = favoriteFood;
        this.Description = $"{Name} is a {breed} and loves to eat {favoriteFood}";
    }
```



```
var sashaTheDog = new Dog("Sasha", "Cheagle", "chicken", Gender.Female);
sashaTheDog.Eat("kibble");
sashaTheDog.Eat("chicken");
sashaTheDog.MakeNoise();
```



```
public Parrot(string name, string featherColor, Gender gender) : base(gender)
{
    this.IsHumanFriendly = true;
    this.PhrasesHeard = new HashSet<string>();
    this.Name = name;
    this.PreferedFood = "seeds";
    this.Wingspan = 12;
    this.FeatherColor = featherColor;
    this.TimesSpoken = 0;
}
```

Practical: Abstraction and Encapsulation

You will make an interface, *IPerson.cs*, which implements *IGenderable, IWalkable,* and *IEatable*

Properties in IPerson:

- Name: String
- Age: int
- Hobbies: HashSet<string>()

Methods in IPerson:

- void Talk(): If the person does not have a hobby, it will print to the console: *I don't have anything to talk about*; if the person has hobbies, it will talk about one of the hobbies
- void AddHoby(string hobby): will add an entry to the set of hobbies

You will then write *Person.cs*, which will implement IPerson!

Advanced OOP: Inheritance and Polymorphism

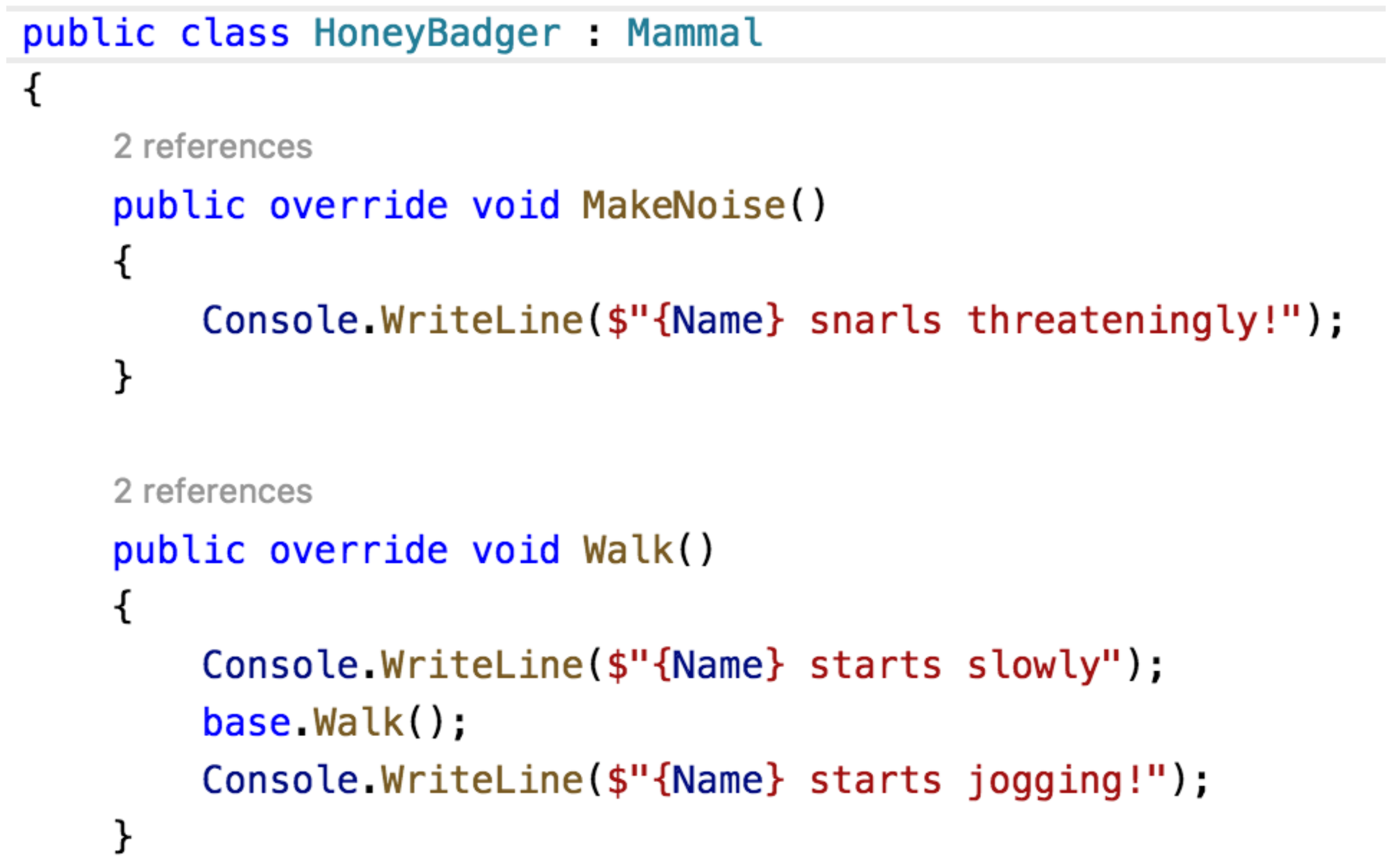
What is inheritance?

Inheritance is the concept of allowing classes to be created that exist "on top" of other classes.

For example, we can make a class for mammals, and have dogs inherit from that class. Often, we will inherit from abstract or partial classes.

Is is often useful to make an abstract class, which is a "partial class", or to use virtual methods when making a class. You can't make a new instance of an abstract class since it's incomplete, but you can inherit from them and build on top of them.

Our Mammal class was an abstract class, and our HoneyBadger class inherits the Mammal class and finishes it up.



```
public class HoneyBadger : Mammal
{
    2 references
    public override void MakeNoise()
    {
        Console.WriteLine($"{Name} snarls threateningly!");
    }

    2 references
    public override void Walk()
    {
        Console.WriteLine($"{Name} starts slowly");
        base.Walk();
        Console.WriteLine($"{Name} starts jogging!");
    }
```

Multiple Inheritance

Interfaces and classes can both inherit from multiple interfaces at a time. This allows us to create small interfaces that describe a very reusable slice of functionality.

For example, our animals all can move! But they all move in different ways.

Rather than declare that all our animals follow the interface rules of IWalkable, or IFlyable, we have each particular abstract class define inherit from IAnimal and whichever movement patterns describe them best.

- Mammal inherits from IWalkable and IAnimal
- Avian inherits from IWalkable, IFlyable, and IAnimal

This can be extended to a school of thought known as *composition over inheritance*

- https://en.wikipedia.org/wiki/Composition_over_inheritance

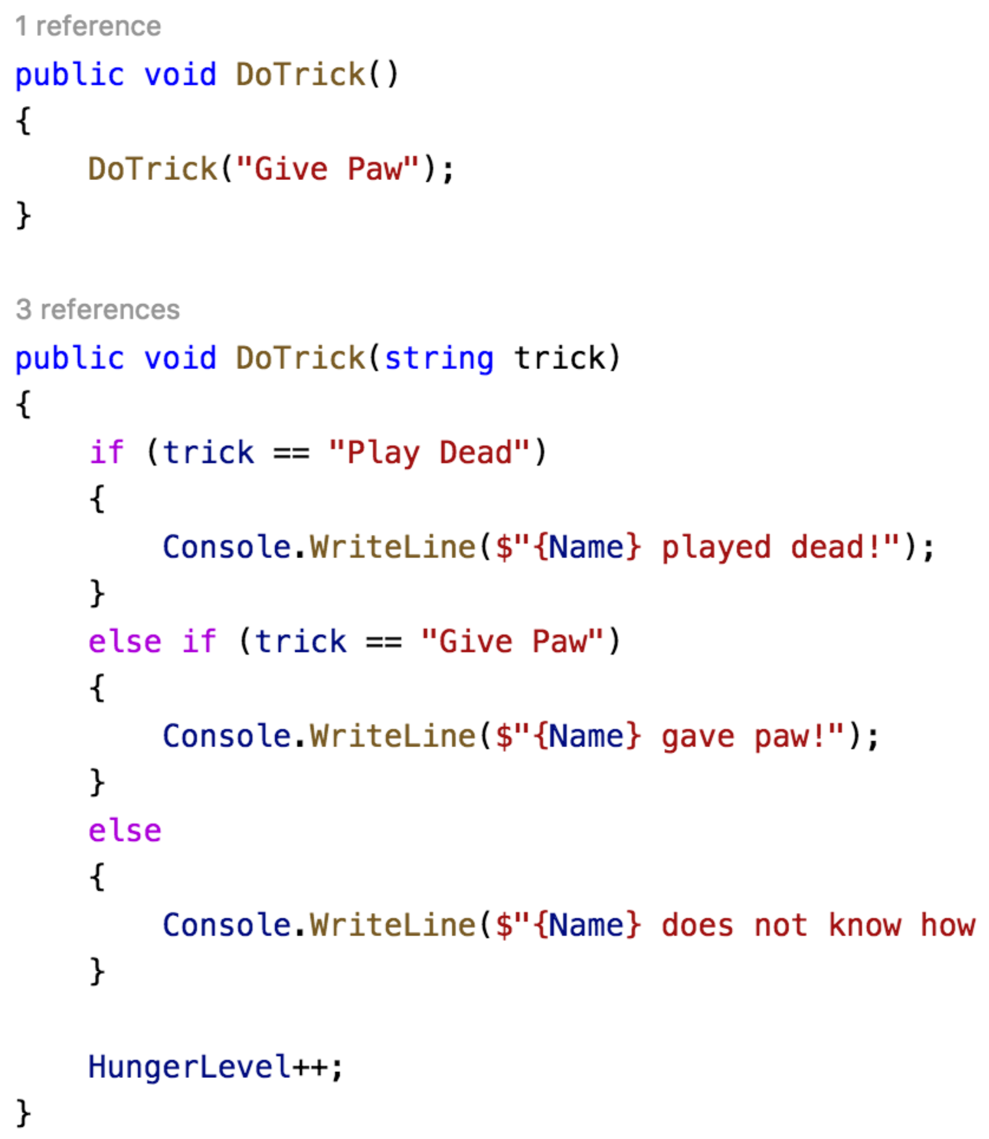
What is polymorphism?

Polymorphism is the concept of your classes using the same name to represent the same action occurring, but in different ways. This comes in two forms: overriding, and overloading

Overloading

Overloading is when a class has a method with the same signature and return type, but different parameters. It is common to see overloads calling other overloads with different parameters.

Here, we see how a Dog has two *DoTrick* overloads: one of them has no particular trick specified, while one of them allows you to specify which trick to have the dog perform.



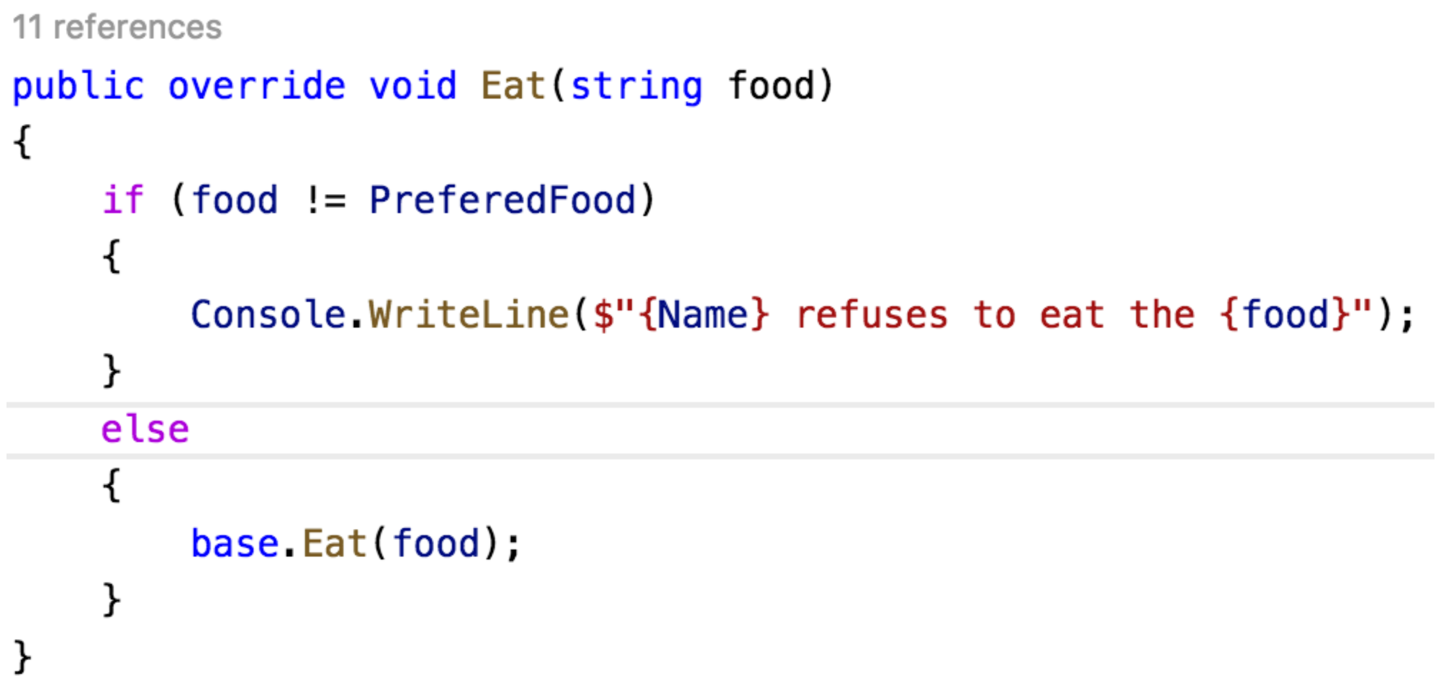
```
1 reference
public void DoTrick()
{
    DoTrick("Give Paw");
}

3 references
public void DoTrick(string trick)
{
    if (trick == "Play Dead")
    {
        Console.WriteLine($"{Name} played dead!");
    }
    else if (trick == "Give Paw")
    {
        Console.WriteLine($"{Name} gave paw!");
    }
    else
    {
        Console.WriteLine($"{Name} does not know how
    }

    HungerLevel++;
}
```

Overriding

Overriding is when a child class makes a method that overrides a parent class's version of the same method.

Even if we override a method on our child class, inside of our class we can actually still call our parent methods by using the *base* reference.

We see this in most of our animals. This demonstration can be found in *Parrot.cs*, where the Parrot will only eat if it's their prefered food.



```
11 references
public override void Eat(string food)
{
    if (food != PreferedFood)
    {
        Console.WriteLine($"{Name} refuses to eat the {food}");
    }
    else
    {
        base.Eat(food);
    }
}
```

Abstract and virtual methods

Abstract methods are when methods are described in a class, but not defined; it's allowing a class to describe the thing that it will need.

Virtual methods are "suggestions" that a class can follow. They can be overridden, or they can be used by their children.

Practical: Putting it all together

Make a new animal, Cat, that inherits from Mammal.

Cats should have a private readonly property called HumanTolerance, which describes how many times it can stand being told to speak before it stops talking and prints the message: "{CATNAME} is bored and does not want to speak"; after being fed, this human toleration is set to the value provided at time of construction

Making the cat class is an example of *abstraction*

Making the cat class have a readonly property called HumanTolerance that it uses internally is *encapsulation*.

Having cat inherit from Mammal is inheritance.

Having the cat override speak and call the parent’s speak method internally is *polymorphism.*