**Inheritance Walkthrough**

Step One: MaleLineage.cs

When you open this up, notice that the class is marked as **abstract**. What this means is that you cannot have an instance/variable of this class directly. Anything that wants to make use of an abstract class has to **inherit** from the abstract class.

Just a few more things to notice before we move on. We also have an **abstract** method defined inside of our **abstract** class. This tells us that the method is being declared here and currently there is no implementation inside of it, but this method must be implemented inside of whatever class inherits from it.

Lastly, we have a method that is marked with the keyword **virtual**. This allows us to declare a method inside of the abstract class that can have implementation already inside of it. That is one of the differences between a **virtual** and **abstract** method, the other difference is that a **virtual** methoddoes not have to be overridden inside of the inheriting class.

Step Two: Father.cs

This is the first class we will see that is inheriting from another object. I say object, because an abstract class is not the only thing we can inherit from. Just after we declare “public class Father” we follow it with a colon and the name of the class we want to inherit from “ : MaleLineage” in this case. After that we have some basic setup: we give the Father class 2 properties (FirstName and LastName) and we set the value of those properties inside of the constructor.

Now let’s take a look at what we can do with those abstract and virtual methods that were declared inside of the abstract class. If we look at YellName and TellJoke, which were our two abstract methods, we see that we now write them as public **override** instead of public **abstract.** This allows us to actually implement the method that was declared. In YellName, we print to the console the name defined in the constructor. And TellJoke prints a joke to the console.

But with our **virtual** method TellDadJoke, we declared that we are going to **override** it but we also added the keyword **sealed**. **Override** allows us to add functionality to the method, but the **sealed** keyword tells this class, and any class that wants to inherit from it that this is the absolute final implementation of this method. It is already the best method it can be and no one beyond this class could possibly have anything more to add to this, aka no one inheriting from Father can **override** TellDadJoke anymore.

And lastly, for the Father class, we can define our own specific methods inside like EmbarassSon which takes in a Son object (we will cover that next) and is able to perform embarrassing actions inside fit for a father with horrible jokes.

Step Three: Son.cs

This time instead of Son also inheriting from MaleLineage, like Father did, we are now inheriting from Father directly. Indirectly, we are also inheriting from MaleLineage because the Father inherited from it. Almost like a grandchild of MaleLineage.

So just like Father did, you can also override YellName and TellJoke in the Son class. This allows Son to make these methods unique to the Son class. But the TellDadJoke is not able to be overridden because it was declared as **sealed** by Father. What you can do though is use the method still because you still inherit the functionality/joke, you just can’t change what it is. So we created a method ActLikeAmySchumer that allows the Son to call TellDadJoke and then add his own twist to it. You can also still call the method TellDadJoke from an instance of the Son class.

One more thing about the use of TellDadJoke. In Son’s constructor, when we wanted to use the property LastName, which was declared inside of the Father class, we used the keyword **base** to explicitly say we wanted to get the value from the Father (because in Father’s constructor the last name is always set to “Skywalker”) Technically, we don’t have to declare base. If the Son class declared its own LastName property, then we would have to explicitly say **base**.

And lastly, we can again make the son more specialized by adding his own flare like fix R2D2 . Of course, the Son also inherited the ability FixC3P0 from Father, but Father doesn’t inherently have the knowledge/ability of how to fix R2D2. So the son is a more specialized version of Father, who is a more specialized/actual implementation of the vision of MaleLineage.

Step Four: Walkthrough Program.cs

Go take a look at the comments in Program.cs and see the stuff we just created in use

Extra Learning:

Difference between inheritance and implementation

What else can you do with an abstract class (we already saw virtual methods and abstract methods)

<https://en.wikipedia.org/wiki/Multiple_inheritance>

Where else can you use the keyword **base**

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/inheritance>

Look up how to have a property named LastName inside of the Son class that is differentiable from the LastName property inside of the Father class