A class that implements an interface can implement an interface method and a method of it’s own with the same name and params by prefixing the method with the interface name:

ISaveable.Save()

{}

This is known as **Explicit implementation**. This results in differing behaviour. If a variable is of the concrete type it will call the concrete type version of the method. If the variable is of type interface it will call the explicit implementation. An explicitly implemented method is hidden from the concrete class.

When we create an explicitly implemented member, we can only call it against the interface.

An object of type interface will only have access to the members of it’s own defined interface even if the interface variable was initialised with a class that has a larger interface. The interface variable won’t have access to the larger class interface.

Why explicit implementation?

When 2 methods have the same name and params but different return types. In C# we’re not allowed to overload based on a change to the return type only.

If a class implements 2 interfaces with the same member name but with differing return values, it will have to implement at least one of them explicitly because C# doesn’t allow overloading based on return value.

When a class implements **IEnumerable**<T> it must also implement **IEnumerable**. They have different return values for getEnumerator(). Basically call the getEnumerator for IEnumerable<T> in the other one and put your real implementation code in getEnumerator for IEnumerable<T>.

We gotta use explicit implementation if we’re implementing more than one interface with the same signature but different return types.