Microsoft Visual Studio 2008

Understanding Object-Oriented Programming

Hands-on Lab

Lab Objective

Estimated time to complete this lab: **60 minutes**

Goals of this activity:

Learn how inheritance works

Understand rules of construction with inheritance

Learn the difference between static and dynamic binding

Learn how to define an interface

Learn how to implement an interface

Learn how to use interfaces to create plug compatible code

Create a delegate

Create a listener class with static and instance methods that match your delegate signature

Invoke the listener functions via your delegate

This lab consists of the following exercises:

Inheritance and Construction
More Inheritance
Defining and implementing an interface
Writing the Client Code
Working only with Interfaces
Starting out

Task 1 – Inheritance and Construction

Create a console application and add a few classes to it.

Criteria:

1. First, set the provided class for your console application to be called Client.
2. Create two other classes, name one "Base" and the other "Derived". In the Base class provide a constructor which takes no arguments. In the implementation of the constructor simply write out to the console "in base constructor".
3. Next, add a method to the Base class called Method1 that returns a void and takes no arguments. In the implementation of this method write out to the console "in

base.method1".
Your Base class should look like the following:

```
class Base
{
public Base()
{
Console.WriteLine("In Base Constructor!");
}
public void Method1()
{
Console.WriteLine("In Base.Method1");
}
}
```

4. Next, add another class called Derived that inherits from base. You don't need write any code in the derived class to test things out.
5. Now add code to the main method in the client class that will declare a reference of of the type Base but instanciate an instance of the type Derived. Then try calling Method1. Be sure and test everything out!
Your main method should look like the following:

```
    Base b;
    b = new Derived();
    //call inherited implementation for method1 on Derived instance.
    b.Method1();
```

6. Compile your application and test.

Notes:
• This is a very simple example using inheritance

Task 2 – More Inheritance

In this section you will simply explore making things slightly more complex by adding another class, more constructors and working with static and dynamic binding.

Criteria:

1. Go modify Method1 in the Base class by marking it "virtual". Then in the

Derived class override the implementation of Method1, making sure to mark the method as "override" in the Derived class. In the implementation of the Derived.Method1 write out to the console "in derived method1". Try and run your application again, you should now see that the new overriden method is called.

2. Now we want to see how the rules of construction work. In the Derived class add a constructor that takes a string as an argument and provide an implementation that writes out to the console. Also, create another class and name it "MoreDerived" and make sure this class inherits from Derived. Once you have done this you will notice that if you try to compile the compiler complains because there is no default constructor in Derived that takes no arguments so the compiler can't generate a constructor for you in MoreDerived and chain the call to Derived because it doesn't know what to pass to the constructor. Fix the MoreDerived class by adding a constructor that and manually chain a call to the Derived constructor. The code for the two classes should look something like the following:

```
class Derived : Base
{
public Derived(string var)
{
            Console.WriteLine("In Derived constructor. Constructor
            parameter value:{0}",var);
}
public override void Method1()
{
Console.WriteLine("In Base.Method1");
}
}
class MoreDerived : Derived
{
   //notice the use of the base keyword to chain the construction to the
   base
public MoreDerived(): base("DEFAULT")
{
Console.WriteLine("In MoreDerived Constructor!");
}
}
```

3. Test out your application by modifying the client to create an instance of more derived and verify that the constructors are called as expected. Your client should look something like the following:

```
Base b = new MoreDerived();
b.Method1();
```

4. Now it's time to look at static binding. Add a method called Method2 that returns a void and takes no arguments to both the Base class and the Derived class, be sure you DO NOT use the virtual and override key words. In the implementations simply write out to the console your class name and method name.

Modify the client class to declare two references, one of the Base type and one of the

Derived type. Instanciate an instance of the derived type and assign the instance to both references. Then using each reference, make a call to each Method2. Your code should look lik the following:

```
class Client
{
static void Main(string[] args)
{
TestStaticBinding();
}
static void TestStaticBinding()
{
Base b;
Derived d;
//create an instance of Derived
d = new Derived("test");
b = d;
b.Method2();//does this go to Derived implementation or Base?
d.Method2();//what about this call?
}
}
class Base
{
public Base()
{
Console.WriteLine("In Base Constructor!");
//where does this call really go?
this.Method1();
//depends on what the client created an instance of!
}
public virtual void Method1()
{
Console.WriteLine("In Base.Method1");
}
//method2 is used to show static binding and the "new" key word
public void Method2()
{
Console.WriteLine("In Base.Method2");
}
}
class Derived : Base
{
public Derived(string var)
{
Console.WriteLine("In Derived constructor. Constructor parameter
value:{0}",var);
}
public override void Method1()
{
Console.WriteLine("In Base.Method1");
}
public new void Method2()
{
```

```
Console.WriteLine("In Derived.Method2");
}
}
```

5. Now test things out. What you should notice is that even though you only have one instance, a derived type, you get two different implementations when you call

Method2 depending on which reference type you use. This happens because the method implementation to call is determine statically at compile time. You will also notice the compiler produces a warning on the method2 suggesting you mark the method in the Derived type with the "new" keyword. This is just ensuring you understand that you are getting static binding not the more desirable dynamic binding.
Next add a line to the base constructor that calls this.Method1(), where will this call go if method1 is virtual? The answer depends on what instance type "this" really points to as it's dynamically bound.

Notes:

• This is a good way to see how inheritance can easily get very complex.

Overview

In this sample you will create a console application that contains 3 classes and 1 interface. One of the classes will be the Client class with a Main entry point, the other two classes will provide different ways to implement the same interface. The point of the two different implementations is to see the advantage of NOT marking your interface implementations public.

Task 3 – Defining and implementing an interface

Working with Interfaces by defining and implementing them.

Criteria:

• Make a new console application by opening VS.NET and chosing "New project" and selecting C# as the language and Console Application for your project template.
• Rename the provided class to be called "Client"

1. Create an interface called IHuman that has one method, Speak that takes a string as a parameter and one property Name as a string. Your interface should look like the following:

```
//declare interface
public interface IHuman
```

```
{
void Speak(string Message);
string Name{get;set;}
}
```

2. Create two classes, name them Theodore and Kirk.

In the Theodore class you will add the implementation by using VS.NET to stub out the method and property signatures for you. To do this simply add the ": IHuman" to the class and then go to the class view from VS.NET, navigate to the Theodore class,Bases and Interfaces and then right click on the IHuman interface and choose "Add" and select "Implement Interface". The VS.NET IDE will provide an empty stub of the interface implementation. Notice that all implementation is public. Provide an actual implementation for the class.

Your Theodore class should look something like the following:

```
public class Theodore : IHuman
{
private string m_Name;
public void Speak(string Message)
{
Console.WriteLine("Hi my name is {0}.\n" + Message, m_Name);
}
public string Name
{
get
{
return m_Name;
}
set
{
m_Name = value;
}
}
}
```

Task 4 – Writing the Client Code

Writting the client code to test out your interface implementations. Yo u will do all your work in the Main method of the Client class for this part.

Criteria:

1. In the Main method add code that declares both a Theodore reference and a IHuman reference.
2. Instanciate an instance of the Theodore class and assign it to the Theodore reference and then add calls to set the name property and make the instance speak. Notice

that you can use a class base reference, ignoring the fact that an interface has even been implemented.

3. Next, assign the instance to the IHuman reference you declared and then make calls to set the name property and speak methods using the interface. This is the goal when using interfaces. Your client code should look something like the following:

```
static void Main(string[] args)
{
IHuman h;
Theodore t;
h = new Theodore();
//cast for assignement
t = (Theodore)h;
//use class based reference
t.Name="Fred";
t.Speak("I like public implementations!");
//use interface reference
h.Name = "Teddy";
h.Speak("I like VB!");
```

Notes:

• You will need to do a cast operation when assigning the interface reference that points to an instance of Theodore to the Theodore reference variable

Task 5 – Working only with Interfaces

Implement the interface on the Kirk class just as you did on the Theodore class, however do not use VS.NET to stub out the implementation!

Criteria:

1. The goal here is to provide an implementation on the class that can only be accessed through the interface! Implement the IHuman interface on the Kirk class but don't use VS.NET to stub out the methods. Instead type them in yourself leaving off the access modifier and ensuring to add the interface prefix to the method/property name. Your class will look like the following:

```
public class Kirk : IHuman
{
private string m_Name;
void IHuman.Speak(string Message)
{
Console.WriteLine(Message);
}
string IHuman.Name
{
get
{
return m_Name;
}
set
{
m_Name = value;
}
}
}
```

2. Next modifiy the client code in Main to work with the interface and the Kirk instance. Be sure to try and work directly with a class based reference and notice that you can't call any of the interface methods with the class reference, only the interface reference.

Notes:

• If you're going to use interfaces it's better to not mark them public so that clients don't end up hard coded against a class based reference!

Overview :

To understand how to create a delegate, implement a method that can be invoked by the delegate and invoke the method through and instance of the delegate.
You will be creating an new Console application project using VS.NET and all of your work will be performed in that one project.

Task 6 – Starting out

Create a new console application project and rename the provided class to be called the Client class. The client class will be used to create an instance of the listener class, the delegate class and tie the two together in order to invoke the delegate.

Criteria:

1. Make a new console application by opening VS.NET and chosing new project and selecting C# as the language and Console Application for your project template.
2. Change the provided class name from Class1 to Client
3. Create a delegate with the following signature

```
delegate void NotifyMe(string sInfo);
```

4. Create a class called Listener and add a static method called GetNotified that matches the signature of the delegate you just defined. Also, create a static method called GetNotifiedAgain that also matches the signature of the delegate. Add some implementation to your events that simply write out the data passed in to the console. Your Listener class should look like the following:

```
class Listener
{
//instance function that matches signature of delegate above
public void GetNotified(string sInfo)
{
Console.WriteLine("I got notified with the following information {0}",sInfo);
}
//static function that matches signature of delegate above
public static void GetNotifiedAgain(string sInfo)
{
Console.WriteLine("I got notified with the following information:{0}",sInfo);
}
}
```

5. Next, modify the Client class by adding a static method called InvokeDelegate that takes a NotifyMe delegate as a parameter. Be sure and invoke the delegate in the implementation. Your method should look like the following:

```
static void InvokeDelegate(NotifyMe d)
{
d("You are late paying your bills!");
}
```

6. Now go to your Client class and add code to Main to create an instance of your delegate, passing the Listener.GetNotifiedAgain method to the constructor and then call the InvokeDelegate function and test your application. Your client code will look something like the following:

```
class Client
{
//Main creates the delegate, points it at a function implementation
//and invokes the delegate
static void Main(string[] args)
{
//use static function of Listener class for delegate //call
//create instance of Notify delegate and point it at static function to call
NotifyMe d = new NotifyMe(Listener.GetNotifiedAgain);
//invoke the delegate function
//notice function being called below takes a delegate //that can point
//to ANY function on any class - this is loosely coupled!
InvokeDelegate(d);
}
```

7. Last, create an instance of your Listener class and another instance of the NotifyMe delegate and try invoking the Instance method of the listener. Your code should look something like the following:

```
class Client
{
//Main creates the delegate, points it at a function implementation
//and invokes the delegate
static void Main(string[] args)
{
//use static function of Listener class for delegate //call
//create instance of Notify delegate and point it at //static function to call
NotifyMe d = new NotifyMe(Listener.GetNotifiedAgain);
//invoke the delegate function
//notice function being called below takes a delegate //that can point
//to ANY function on any class - this is loosely coupled!
InvokeDelegate(d);
//invoke delegate using Instance function
//create listener instance
Listener lsnr = new Listener();
//create delegate and point to listener instance method
NotifyMe d2 = new NotifyMe(lsnr.GetNotified);
//invoke just like before
InvokeDelegate(d2);
}
}
```

Notes:

- This sample assumes you are a little familiar with VS.NET