Module 04 Notes

Abstraction, Inheritance, and Polymorphism

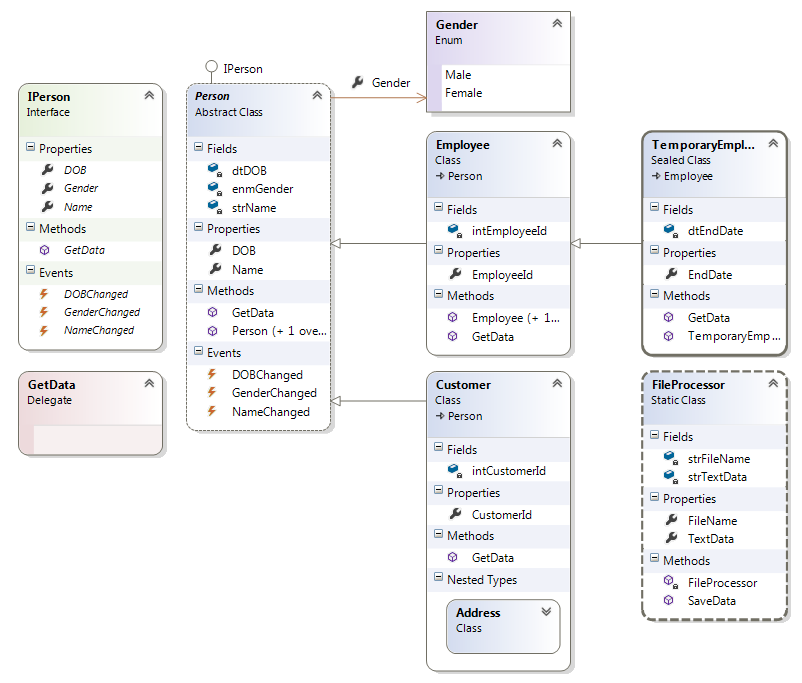
There are three basic principles adhered to with programing modern application using Object Oriented Programming (OOP); Abstraction (Encapsulation), Inheritance, and Polymorphism.

In this module we will demonstrate how these are used.

# Abstraction

With abstraction, you design your applications so that the underlying data and processing is not worked with directly. So, variables and methods are only accessed though some abstraction layer.

* Variables are accessed through Properties
* Methods are accessed through Delegates
* Classes or Structures are accessed through Interfaces



Note that in this example, a variable of type IPerson could be used to point to an Employee, Customer, or TemporaryEmployee object. While, a variable of the delegate type GetData could be used to point to any method with the same Signature and Return Type. Each variable working with objects in the “abstract.”

# Encapsulation

Encapsulation is closely related to Abstraction, but it is a tool used to create abstractions rather than an actual principal of OOP programming.

With encapsulation, you wrap your programming structures into a capsule of code and use these structures in the “abstract.”

* Property procedures are way of encapsulation code to work with field variables and constants
* Method are a way to encapsulation groups of statements
* Classes and Structures are a way to encapsulation Properties and Methods

# Inheritance

The concept of inheritance asks you to program your applications so that general encapsulation code structures (classes) will pass its code on to more specific encapsulation code structures. So a parent class passes code to a child class which in turn can pass the combined code to a further child class.

* Abstract classes pass code to Concrete classes
* Concrete Parent classes pass code to Concrete child classes

## Inherited Constructors

Unless you say otherwise, the DEFAULT constructor of a base class is IMPLICITLY called before the logic of the derived constructor is executed. To optimize the creation of a derived class EXPLICITLY call an appropriate custom base class constructor.

public Employee(): **base**()

{ }

public Employee(int EmployeeId, string Name, DateTime DOB, Gender Gender): base(Name, DOB, Gender)

{

this.EmployeeId = EmployeeId;

// this.Name = Name; **This** is not needed since the base constructor will be

// passed the data in with :base(Name, DOB, Gender)

}

## Inheritance Modifiers

In .NET there are three types of Inheritance Modifiers

* Abstract
  + Requires inheritance before you can make and object instance.
* Static
  + Prevents creating object instances, you must use the class directly
* Sealed
  + Prevents inheritance, you can only make an object instance
  + Sealed classes cannot be extended, but it is possible to reuse the functionality found within a class marked sealed with containment/delegation model (I.E. by making an object through another class)

# Polymorphism

The concept Polymorphism asks you to program your applications so that one encapsulation code can have many versions of itself.

* One Abstract or Parent class can have many child classes
* One Method can have many versions of itself with different signatures
* One Interface can be implemented by many Classes and Structures

# Making Code Reusable

The main desired outcome of using the principles of OOP is to create reusable components of code. This code comes in two flavors:

1. Classic Inheritance = "is-a" relationship
   1. Parent - Child Classes
   2. Base classes define general characteristics of all descendants
   3. The .NET platform does not allow multiple inheritance for classes
2. Containment/delegation model = "has-a" relationship
   1. Person Class has a Gender object (delegation)
   2. Customer Class as a nested Address Class (Containment)

Your instructor will now review the five basic types found in the OOPPrinciples project.

# Lab 04-1

In this lab, you will create an example of an application designed along the principals of OOP. First you will create examples of the five basic types and then you will test your work using a simple windows form application for a Test Harness.

1. Create a new Windows project called MyOOPDemo in a new solution called Mod04Labs.
2. Create an Interface called IRoom using the VS diagram tool in the new project.

* Fields: strNumber, fltWidth, fltSize, intOutlets
* Constructor: Room(), Room(Number, Width, Size, Outlets)
* Properties: Number, Width, Size, Outlets
* Methods: GetData()

1. Create an abstract class called Room that implements the IRoom interface.
2. Add the following to the Room class:
3. Create a child class called Classroom that inherits from the Room class.
4. Add the following to the Room class:

* Fields: blnHasProjector, blnHasWhiteboard
* Constructor: Classroom(), Classroom(Number, Width, Size, Outlets, HasProjector, HasWhiteboard )
* Properties: Number, Width, Size, Outlets, HasProjector, HasWhiteboard
* Methods: GetData()

1. Create a child class called Office that inherits from the Room class.
2. Add the following to the Office class:

* Fields: blnHasWindow
* Constructor: Office(), Office(Number, Width, Size, Outlets, HasWindow)
* Properties: Number, Width, Size, Outlets, Projector, HasWindow
* Methods: GetData()

1. Add code to Form1 that allows you to test that new objects can be created using the Classroom and Office classes but referenced by variables of both IRoom interface and the Room abstract class.
2. Make comments explain which OOP principles are being used in your code and where.

Estimated Time: 20 Minutes

# Programming for Delegation

(Not the same as a Delegate type!)

Delegation defines the roles of objects instead of the relationships between classes. These objects are said to represent a “Has-a” relationship.

Here is an example that shows an Employee objects that **HAS-A** BenefitPackage object:

public partial **class Employee**{

// **USES** a BenefitPackage object.

protected BenefitPackage empBenefits = **new BenefitPackage**();

// **Expose** certain benefit **behaviors** of object.

public double **GetBenefitCost**()

{ return empBenefits.**ComputePayDeduction**(); }

// **Expose** object through a custom property.

public **BenefitPackage** Benefits {

get { return empBenefits; }

set { empBenefits = value; }

}

…

In this example, to use a BenefitPackage object you must work with it though an Employee object.

Delegation’s meaning is often debated in the industry, but it practical use in .NET is by including the following techniques in your applications:

* Indirectly expose the functionality of an object (made from one of the 5 basic types) with properties and methods, thereby forcing its use in the abstract.
* Add public members to a containing Class or Structure that make use of a contained object's functionality.

**Note:** Inheritance represents an “is – a” relationship while delegation represents a “has -a” relationship.

# Programming for Containment

Containment is used to group data and behaviors. These behaviors are not usually exposed to code outside of the container (Class or Structure). In .NET application this takes the form of Nested Class or Structures.

Here is an example:

public class OuterClass {

// A public nested type can be used by anybody.

**public** class PublicInnerClass { }

// A PRIVATE NESTED TYPE can only be used by members of the containing class.

**private** class PrivateInnerClass { }

}

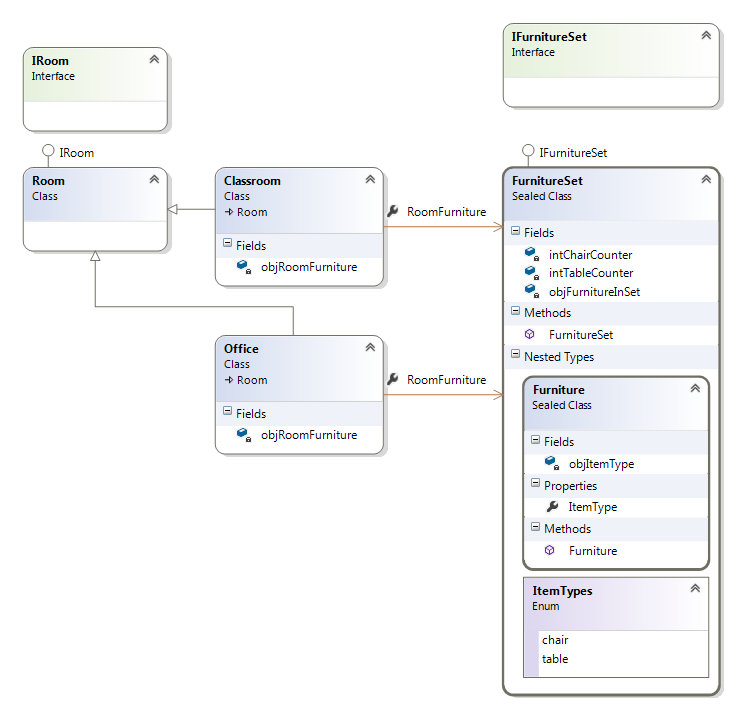
The following are traits of nesting a type:

* Nested types define a enum, class, interface, struct, or delegate directly within the scope of a class or structure
* Nested types are used to organize private code
* Nested types allow you to gain complete control over the access level of the inner type, as they may be declared privately *(non-nested classes cannot be declared using the private keyword*)
* Because a nested type is a member of the containing class, it can access private members of the containing class
* Oftentimes, a nested type is only useful as a helper for the outer class, and is not intended for use by the outside world
* Nested types are not usually exposed to the outside objects

# Lab 04-2

In this lab, you will modify your application to include an example of Delegation

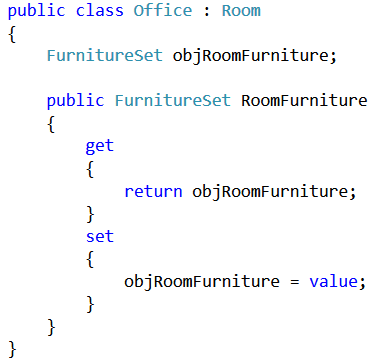
1. Create an Interface called IFurnitureSet and a Class called FurnitureSet
2. Create a nested Class called Furniture and an Enum called ItemTypes



1. Add the following code:



1. Add code to the Classroom and Office classes that allow furniture sets to be added and reviewed (set/get).



1. Use a button to create a test object and walk through the object creation use the VS debugger.
2. Make comments explain which Containment or Delegate principles are being used in your code and where.

Estimated Time: 20 Minutes