**Encapsulation in C#**

Often considered the first pillar of object-oriented programming, encapsulation can be used to describe the accessibility of the members belonging to a class or struct.  C# provides access modifiers and properties to help implement encapsulation in your classes.  While some consider this accessibility configuration to be the only aspect of encapsulation, others also define encapsulation as the act of including all data and behavior required of the class, within the class definition.   This definition can be stretched a bit in C# when using partial classes.  Review the topic on classes to see what partial classes are all about.

**Private vs Public vs Protected vs Internal**

The following table discusses the access modifiers that can be applied to class members to control how they can be accessed by other code in the application. This is a part of encapsulation by allowing you to restrict access to members where it makes sense.

|  |  |
| --- | --- |
| **Access modifier** | **Description** |
| public | The type is available to code running in any assembly that references the assembly in which the class is contained. |
| internal | The type is available to any code within the same assembly, but not available to code in another assembly. |
| private | The type is only available to code within the class that contains it. You can only use the private access modifier with nested classes. This is the default value if you do not specify an access modifier. |
| protected | The type is only accessible within its class and by derived class instances. |

The tradition is to create private data fields in the class to prevent direct manipulation of the values for those fields, and expose properties to provide access to the values indirectly.  The properties are known as accessors or getters and setters.

**Properties**

As a part of encapsulation, you should consider using properties in your class files. Properties enable you to permit users of the class a means of getting and setting values for the private member data fields within your class.  Properties accomplish while hiding implementation or verification code that you may have written inside the property.   For example, you may want to validate a birthdate that has been passed in to ensure it is in the proper format or that it is in the correct range for the application's usage.  Setting your member variables to private is known as a form of data hiding.  Some also consider data hiding to be part of encapsulation.

Properties also present an "interface" to your class by exposing a way to get or set the members of the class that the user can trust.   In other words, if you have a property called public void Birthdate(date birth), that accepts a birthdate from a user, you can implement the validation code in anyway you see fit, such as using regular expressions to validate or perhaps some custom logic to verify the date range, and then later change that validation logic without impacting the use of the property.  Users still just pass in a birthdate in date format.

The following code shows an example of properties being declared in the DrinksMachine class:  
  
public class DrinksMachine  
{  
   // private member variables  
   private int age;  
   private string make;  
  
  
   // public properties   
   public int Age   
   {   
      get  
      {  
         return age;  
      }  
      set  
      {   
         age = value;   
      }  
   }  
   public string Make  
   {   
      get  
      {  
         return make;  
      }  
      set  
      {   
         make = value;   
      }  
   }  
     
   // auto-implemented property  
   public string Model { get; set; }  
  
      // Constructors  
   public DrinksMachine(int age)  
   {  
      this.Age = age;  
   }  
   public DrinksMachine(string make, string model)  
   {  
      this.Make = make;  
      this.Model = model;  
   }  
   public DrinksMachine(int age, string make, string model)  
   {  
      this.Age = age;  
      this.Make = make;  
      this.Model = model;  
   }  
}  
  
The properties are Age, Make, and Model.   These properties would be backed by private member variables called age, make, and model.

**Property Types**

You can create two basic types of properties in a C# class.  Read only or read-write: (Technically you can also create a write-only property but that is not common.

* A get property accessor is used to return the property value
* A set accessor is used to assign a new value.  (Omitting this property makes it read only)
* A value keyword is used to define the "value" being assigned by the set accessor.
* Properties that do not implement a set accessor are read only.
* For simple properties that require no custom accessor code, consider the option of using auto-implemented properties.

Auto-implemented properties make property-declaration more concise when creating simple accessor methods (getter and setter). They also enable client code to create objects. When you declare a properties this way, the compiler will automatically create a private, anonymous field in the background that can only be accessed through the get and set accessors.  
   
The following example demonstrates auto-implemented properties:  
  
    // Auto-implemented properties   
    public double TotalPurchases { get; set; }  
    public string Name { get; set; }  
    public int CustomerID { get; set; }

**Using Constructors**  
  
If you take a look at the topic on Creating Classes, you'll notice that instantiate a class we used this line of code:

DrinksMachine dm = new DrinksMachine();

Notice how this looks similar to the syntax for calling a method. This is because when you instantiate a class, you are actually calling a special method called a constructor. A constructor is a method in the class that has the same name as the class. Constuctors do not use a return value however, not even void, and they must have the same name as the class file.

Constructors are often used to specify initial or default values for data members within the new object, as shown by the following example:

// Adding a Constructor  
public class DrinksMachine  
{  
   public int Age { get; set; }  
   public DrinksMachine()  
   {  
      Age = 0;  
   }  
}

A constructor that takes no parameters is known as the default constructor. This constructor is called whenever someone instantiates your class without providing any arguments. If you do not include a constructor in your class, the Visual C# compiler will automatically add an empty public default constructor to your compiled class.

In many cases, it is useful for consumers of your class to be able to specify initial values for data members when the class is instantiated. For example, when someone creates a new instance of DrinksMachine, it might be useful if they can specify the make and model of the machine at the same time. Your class can include multiple constructors with different signatures that enable consumers to provide different combinations of information when they instantiate your class.  Recall method overloading.

The following example shows how to add multiple constructors to a class:

// Adding Multiple Constructors  
public class DrinksMachine  
{  
   public int Age { get; set; }  
   public string Make { get; set; }  
   public string Model { get; set; }  
   public DrinksMachine(int age)  
   {  
      this.Age = age;  
   }  
   public DrinksMachine(string make, string model)  
   {  
      this.Make = make;  
      this.Model = model;  
   }  
   public DrinksMachine(int age, string make, string model)  
   {  
      this.Age = age;  
      this.Make = make;  
      this.Model = model;  
   }  
}

Consumers of your class can use any of the constructors to create instances of your class, depending on the information that is available to them at the time. For example:

// Calling Constructors  
var dm1 = new DrinksMachine(2);  
var dm2 = new DrinksMachine("Fourth Coffee", "BeanCrusher 3000");  
var dm3 = new DrinksMachine(3, "Fourth Coffee", "BeanToaster Turbo");