

Module 05(OOP)

Creating Classes and Members

In Visual C#, you can define your own custom types by creating classes. As a programming construct, the class is central to object-oriented programming in Visual C#. It enables you to encapsulate the behaviors and characteristics of any logical entity in a reusable and extensible way. In this lesson, you will learn how to create, use, and test classes in your own applications.

In Visual C#, a class is a programming construct that you can use to define your own custom types. When you create a class, you are effectively creating a blueprint for the type. The class defines the behaviors and characteristics, or class members, which are shared by all instances of the class. You represent these behaviors and characteristics by defining methods, fields, properties, and events within your class.

Suppose you create a class named *DrinksMachine*.

You use the class keyword to declare a class, as shown in the following example:

```
//Declaring a Class
public class DrinksMachine
{
    // Methods, fields, properties, and events go here.
}
```

The class keyword is preceded by an access modifier, such as *public* in the above example, will be described in the Encapsulation section.

Adding Members to a Class

You would use fields and properties to define the characteristics of a drinks machine, such as the make, model, age, and service interval of the machine. You would create methods to represent the things that a drinks machine can do, such as make an espresso or make a cappuccino. Finally, you would define events to represent actions that might require your attention, such as replacing coffee beans when the machine has run out of coffee beans.

Within your class, you can add methods, fields, properties, and events to define the behaviors and characteristics of your type, as shown in the following example:

```
// Defining Class Members
public class DrinksMachine
{
    // The following statements define a property with a private field.
```

```

private string _location;
public string Location
{
    get
    {
        return _location;
    }
    set
    {
        if (value != null)
            _location = value;
    }
}
// The following statements define properties.
public string Make {};
public string Model {};
// The following statements define methods.
public void MakeCappuccino()
{
    // Method logic goes here.
}
public void MakeEspresso()
{
    // Method logic goes here.
}
// The following statement defines an event. The delegate definition is not shown.
public event OutOfBeansHandler OutOfBeans;
}

```

Partial Classes

C# can also implement partial classes. Partial classes allow you to split the definition of the class across multiple source files. Then you compile your application, all of the parts are combined into a single file.

Partial classes are useful when:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on the same class at the same time.
- When working with automatically generated source. Visual Studio uses this approach when your application uses Windows Forms, Web service wrapper code, etc. Microsoft recommends

that you do not modify the auto-generated code for these components as it could be overwritten when the application is compiled or the project files changed. Instead, you can create another portion of the class, as a partial class with the same name, and make your additions and edits there.

An example of using partial classes follows:

```
public partial class DrinksMachine
{

    public void MakeCappuccino()
    {
        // Method logic goes here.
    }
}
```

```
public partial class DrinksMachine
{

    public void MakeEspresso()
    {
        // Method logic goes here.
    }
}
```

Note: you can also split structs and interfaces across multiple source files as well.

Instantiating Classes

A class is just a blueprint for a type. To use the behaviors and characteristics that you define within a class, you need to create instances of the class. An instance of a class is called an object.

To create a new instance of a class, you use the new keyword, as shown in the following example:

```
// Instantiating a Class
DrinksMachine dm = new DrinksMachine();
```

When you instantiate a class in this way, you are actually doing two things:

- You are creating a new object in memory based on the DrinksMachine type.

- You are creating an object reference named `dm` that refers to the new `DrinksMachine` object.

When you create your object reference, instead of explicitly specifying the `DrinksMachine` type, you can allow the compiler to deduce the type of the object at compile time. This is known as type inference. To use type inference, you create your object reference by using the `var` keyword, as shown in the following example:

```
// Instantiating a Class by Using Type Inference
var dm = new DrinksMachine();
```

In this case, the compiler does not know in advance the type of the `dm` variable. When the `dm` variable is initialized as a reference to a *DrinksMachine* object, the compiler deduces that the type of `dm` is `DrinksMachine`. Using type inference in this way causes no change in how your application runs, it is simply a shortcut for you to avoid typing the class name twice. In some circumstances, type inference can make your code easier to read, while in other circumstances it may make your code more confusing. As a general rule, consider using type inference when the type of variable is absolutely clear.

After you have instantiated your object, you can use any of the members—methods, fields, properties, and events—that you defined within the class, as shown in the following example:

```
// Using Object Members
var dm = new DrinksMachine();
dm.Make = "Fourth Coffee";
dm.Model = "Beancrusher 3000";
dm.Age = 2;
dm.MakeEspresso();
```

This approach to calling members on an instance variable is known as dot notation. You type the variable name, followed by a period, followed by the member name. The IntelliSense feature in Visual Studio will prompt you with member names when you type a period after a variable.

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. Constructors 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");
```

Creating Static Classes and Members

In some cases, you may want to create a class purely to encapsulate some useful functionality, rather than to represent an instance of anything. For example, suppose you wanted to create a set of methods that convert imperial weights and measures to metric weights and measures, and vice versa. It would not make sense if you had to instantiate a class in order to use these methods, because you do not need to store or retrieve any instance-specific data. In fact, the concept of an instance is meaningless in this case.

In scenarios like this, you can create a static class. A static class is a class that cannot be instantiated. To create a static class, you use the `static` keyword. Any members within the class must also use the `static` keyword, as shown in the following example:

```
// Static Classes
public static class Conversions
{
    public static double PoundsToKilos(double pounds)
    {
        // Convert argument from pounds to kilograms
        double kilos = pounds * 0.4536;
        return kilos;
    }
    public static double KilosToPounds(double kilos)
    {
        // Convert argument from kilograms to pounds
        double pounds = kilos * 2.205;
        return pounds;
    }
}
```

To call a method on a static class, you call the method on the class name itself instead of on an instance name, as shown by the following example:

```
//Calling Methods on a Static Class
double weightInKilos = 80;
double weightInPounds = Conversions.KilosToPounds(weightInKilos);
```

Static Members

Non-static classes can include static members. This is useful when some behaviors and characteristics relate to the instance (instance members), while some behaviors and characteristics relate to the type itself (static members). Methods, fields, properties, and events can all be declared static. Static properties are often used to return data that is common to all instances, or to keep track of how many instances of a class have been created. Static methods

are often used to provide utilities that relate to the type in some way, such as comparison functions.

To declare a static member you use the static keyword before the return type of the member, as shown by the following example:

```
// Static Members in Non-static Classes
public class DrinksMachine
{
    public int Age { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
    public static int CountDrinksMachines()
    {
        // Add method logic here.
    }
}
```

Regardless of how many instances of your class exist, there is only ever one instance of a static member. You do not need to instantiate the class in order to use static members. You access static members through the class name rather than the instance name, as shown by the following example:

```
// Access Static Members
int drinksMachineCount = DrinksMachine.CountDrinksMachines();
```

Anonymous classes

As you might expect, an anonymous class is a class that does not have a name. Anonymous classes offer the programmer a convenient way of encapsulating read-only properties into a single object without the need to explicitly define a type first. The type name will be generated by the compiler. The type name is also not available at the source code level and the type of each property included in this anonymous class will be inferred by the compiler.

To create an anonymous class, you simply use the new keyword followed by a pair of braces to define fields and values for the class. The following is an example:

```
anAnonymousObject = new { Name = "Tom", Age = 65 };
```

The class will have two public fields, Name (initialized to the string "Tom") and Age (initialized to 65). The compiler has inferred the types of these two fields based on the types of data you initialize them with.

Because our anonymous class doesn't have a name, how can you create an object of that type and assign an instance of the class to it? In the preceding code example, what should the type of the object variable `anAnonymousObject` be? As a result of the way anonymous classes work, you don't know, which is precisely the point of anonymous classes.

This doesn't truly present a problem however, as long as you declare `anAnonymousObject` as an implicitly typed variable by using the `var` keyword as shown here:

```
var anAnonymousObject = new { Name = "Tom", Age = 65 };
```

Recall that using the `var` keyword will result in the compiler creating a variable using the same type as the expression that was used to initialize it. In this case, the type of the expression is whatever name the compiler happens to generate for the anonymous class.

Once instantiated, you can access the fields in the object by using dot notation, as shown in this example:

```
Console.WriteLine("Name: {0} Age: {1}", anAnonymousObject.Name,  
anAnonymousObject.Age);
```

Once created, you have the option to create other instances of the same anonymous class but with different values:

```
var secondAnonymousObject = new { Name = "Hal", Age = 46 };
```

The C# compiler will look at the names, types, number, and the order of the fields in the object in order to determine whether two instances of an anonymous class have the same type or not. In our two examples, both objects contain the same number of fields, the same name and the same type, in the same order. As a result, both variables are instances of the same anonymous class. This means that you can assign `anAnonymousObject` to the `secondAnonymousObject` or vice versa:

```
secondAnonymousObject = anAnonymousObject;
```

Note: There are quite a few restrictions on the contents of an anonymous class:

- anonymous classes can contain only public fields
- the fields must all be initialized
- fields cannot be static
- you cannot define any methods for them

Module Five Assignment

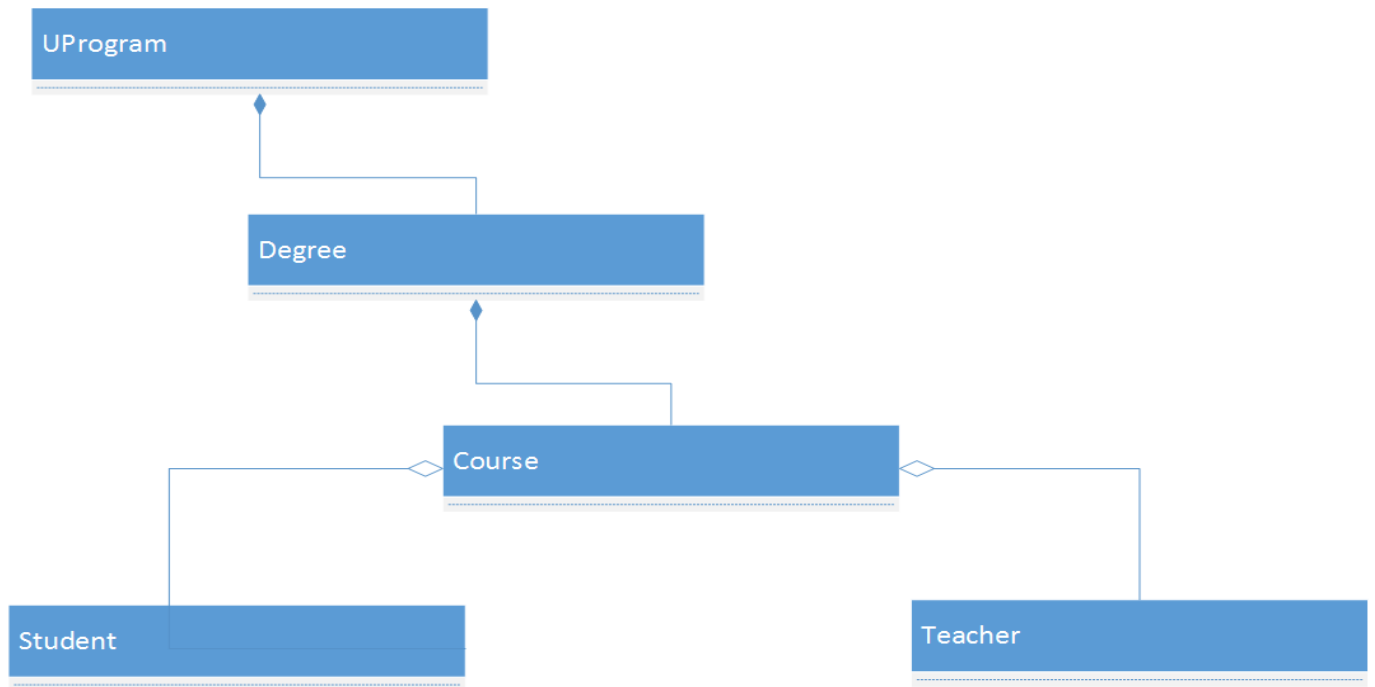
In this assignment, you need to convert your objects for the application into class files. Create a class file for:

- A Student
- A Teacher
- A UProgram (*C# uses Program as the name of the class that contains Main() so we must use a different class name here*)
- A Degree
- A Course

Transfer the variables you created in Module 1 into these class files. Ensure that you encapsulate your member variables in the class files using properties.

The Course object should contain an array of Student objects so ensure that you create an array inside the Course object to hold Students as well as an array to contain Teacher objects as each course may have more than one teacher or TAs. For this assignment, create arrays of size 3 for students and the same for teachers. The UProgram object should be able to contain degrees and the degrees should be able to contain courses as well but for the purposes of this assignment, just ensure you have a single variable in UProgram to hold a Degree and single variable in Degree to hold a Course.

Use this diagram as an example of how the objects relate to each other.

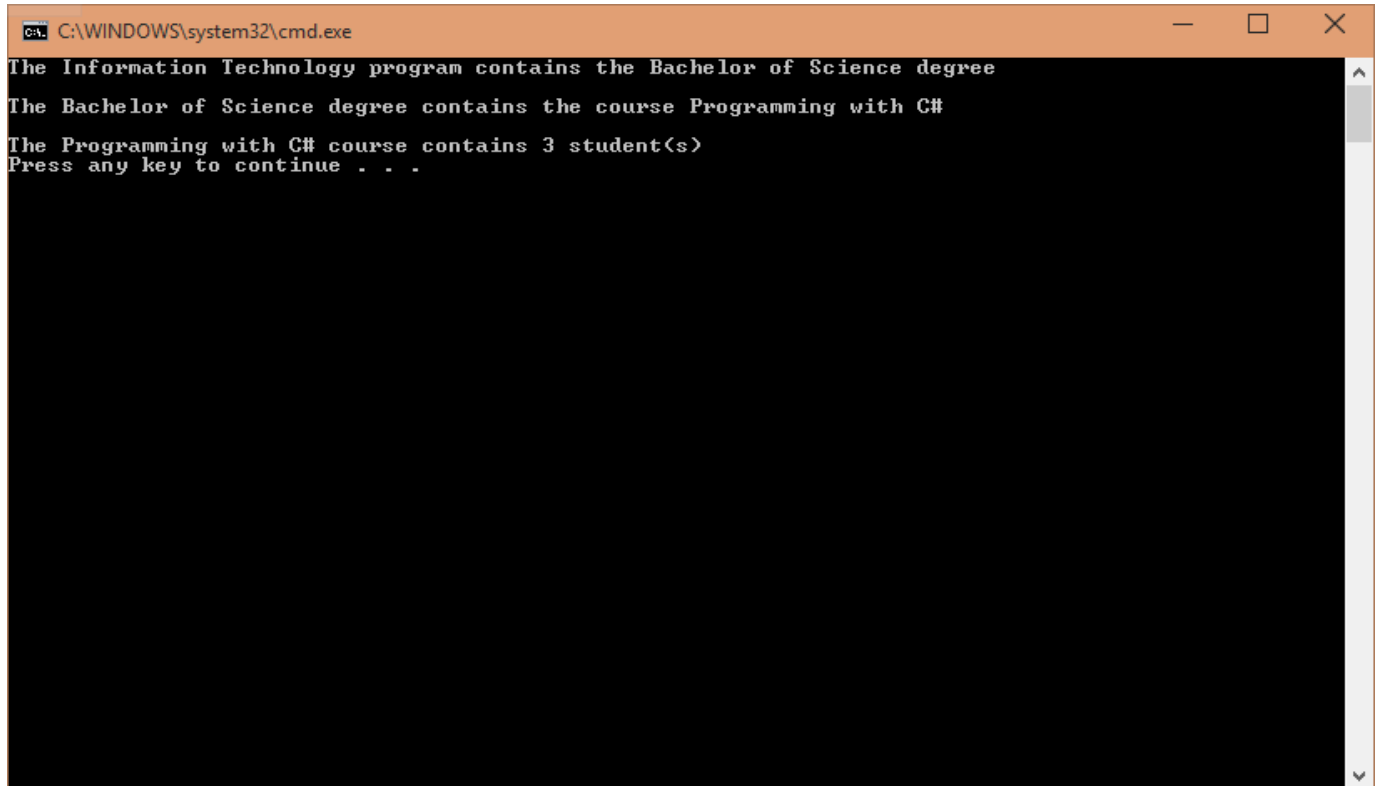


Add a static class variable to the Student class to track the number of students currently enrolled in a school. Increment a student object count every time a Student is created.

In the Main() method of Program.cs:

1. Instantiate three Student objects.
2. Instantiate a Course object called **Programming with C#**.
3. Add your three students to this Course object.
4. Instantiate at least one Teacher object.
5. Add that Teacher object to your Course object
6. Instantiate a Degree object, such as **Bachelor**.
7. Add your Course object to the Degree object.
8. Instantiate a UProgram object called **Information Technology**.
9. Add the Degree object to the UProgram object.
10. Using Console.WriteLine statements, output the following information to the console window:
 1. The name of the program and the degree it contains

2. The name of the course in the degree
3. The count of the number of students in the course.
4. Your output should look similar to this:



A screenshot of a Windows command prompt window. The title bar is orange and shows the path 'C:\WINDOWS\system32\cmd.exe' with standard window controls. The black command prompt area contains the following text in white: 'The Information Technology program contains the Bachelor of Science degree', 'The Bachelor of Science degree contains the course Programming with C#', 'The Programming with C# course contains 3 student(s)', and 'Press any key to continue . . .'. A vertical scrollbar is visible on the right side of the command prompt area.

```
C:\WINDOWS\system32\cmd.exe
The Information Technology program contains the Bachelor of Science degree
The Bachelor of Science degree contains the course Programming with C#
The Programming with C# course contains 3 student(s)
Press any key to continue . . .
```