

Systems Development: Object Oriented Programming

(H172 35)

Introducing Classes

GradeBook Walkthrough

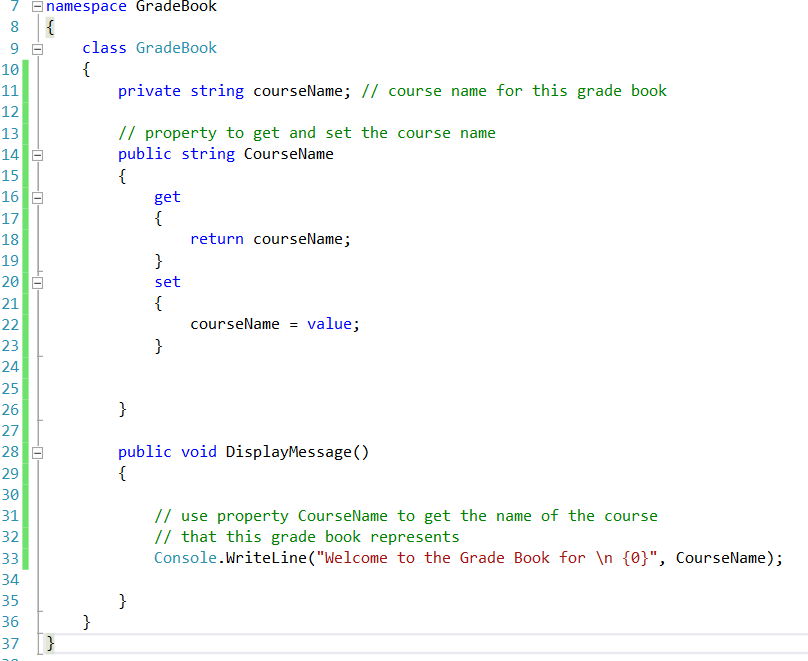
Step 3 – instance variables and properties

An object has attributes that are carried with it as it’s used in an app. Such attributes exist before a method is called on an object and after the method completes execution. Attributes are represented as variables in a class declaration. Such variables are called fields and are declared inside a class declaration but outside the bodies of the class’s method declarations. When each object of a class maintains its own copy of an attribute, the field that represents the attribute is also known as an instance variable—each object (instance) of the class has a separate instance of the variable.

A class normally contains one or more **properties** that manipulate the attributes that belong to a particular object of the class, rather than manipulating them directly. When using the Visual Studio IDE we go to the Properties window to get access to properties which are used to manipulate an object’s attributes. For example, we use a Label’s Text property to specify the text to display on the Label. Now we are going to use a property in code rather than in the Properties window of the IDE. To do this, we first declare a property as a member of the GradeBook class.The example in this walkthrough demonstrates a GradeBook class that contains a courseName instance variable to represent a particular GradeBook object’s course name, and a CourseName property to manipulate courseName.

As you’ll soon see, the GradeBook’s CourseName property can be used to **store** a course name in a GradeBook (in instance variable courseName) or **retrieve** the GradeBook’s course name (from instance variable courseName).

Update the GradeBook class in GradeBook.cs to define an instance variable courseName and a property CourseName for the variable as follows:



Line 11 is a declaration for an instance variable, because the variable is declared in the class’s body (lines 9–36) but outside the bodies of the class’s method (lines 28–35) and property (lines 14–26). Every instance (i.e., object) of class GradeBook contains one copy of each instance variable.

All the methods and properties of class GradeBook can directly manipulate its instance variable courseName, but it’s considered good practice for methods of a class to use that class’s properties to manipulate instance variables (as we do in line 33 of method DisplayMessage).

Most instance-variable declarations are preceded with the keyword private (as in line 11). Like public, keyword private is an access modifier. Variables, properties or methods declared with access modifier private are accessible only to members (such as properties and methods) of the class in which they’re declared. Thus, variable courseName can be used only in property CourseName and method DisplayMessage of class GradeBook.

Declaring instance variables with access modifier private is known as information

hiding (or encapsulation). When an app creates (instantiates) an object of class

GradeBook, variable courseName is encapsulated (hidden) in the object and can be accessed only by members of the object’s class.

How can we allow a program to manipulate a class’s private instance variables but ensure that they remain in a valid state? We need to provide controlled ways for programmers to “get” (i.e., retrieve) the value in an instance variable and “set” (i.e., modify) the value in an instance variable. Although you can define methods like GetCourseName and SetCourse-Name, C# properties provide a more elegant solution.

The GradeBook class’s CourseName property declaration is located in lines 14–26. The property begins in line 14 with an access modifier (in this case, public), followed by the type that the property represents (string) and the property’s name (Course-Name). Properties use the same naming conventions as methods and classes and by convention, we name each property with the capitalized name of the instance variable that it manipulates. Properties contain accessors that handle the details of returning and modifying data. A property declaration can contain a get accessor, a set accessor or both. The get accessor (lines 16–19) enables a client to read the value of private instance variable courseName; the set accessor (lines 20–23) enables a client to modify courseName.

After defining a property, you can use it like a variable in your code. For example, you can assign a value to a property using the = (assignment) operator. This executes the property’s **set** accessor to set the value of the corresponding instance variable. Similarly, referencing the property to use its value (for example, to display it on the screen) executes the code in the property’s **get** accessor to obtain the corresponding instance variable’s value.

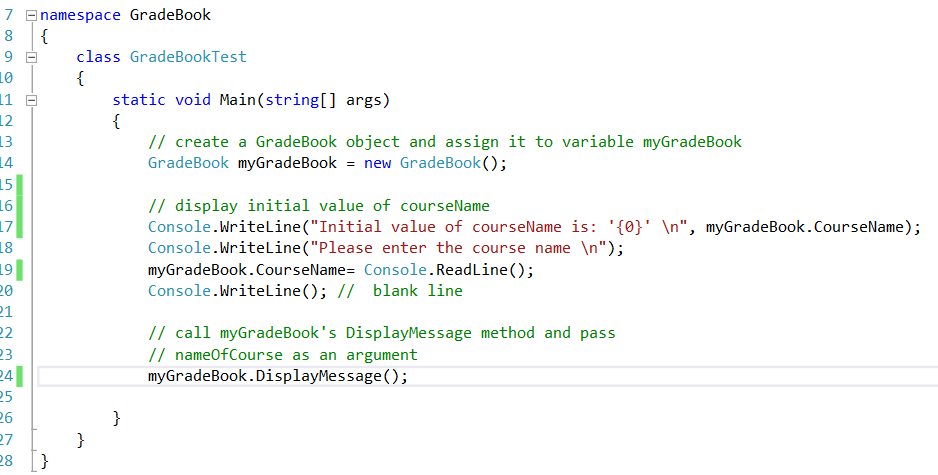
The set accessor (lines 20–23) begins with the identifier set and its body is delimited by braces. When the property CourseName appears in an assignment statement, as in

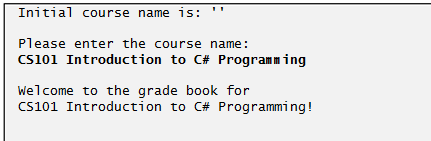
gradeBook.CourseName = "CS100 Introduction to Computers";

the text "CS100 Introduction to Computers" is assigned to the set accessor’s *contextual (reserved)* keyword named **value** and the set accessor executes. Note that **value** is implicitly declared and initialized in the set accessor—it’s a compilation error to declare a local variable value in this body. Line 22 stores the contents of value in instance variable courseName. A set accessor does not return any data when it completes its task.

Method DisplayMessage (lines 28–35) does not receive any parameters. Line 33 outputs a welcome message that includes the value of instance variable courseName. We do not reference courseName directly. Instead, we access property CourseName, which executes the property’s get accessor, returning the value of courseName.

We are now going to update the client code, GradeBookTest.cs to set he courseName variable of the GradeBook object it creates. Modify the GradeBookTest.cs code as below:



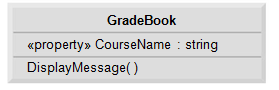


Line 17 displays the initial course name using the object’s CourseName property, this executes the property’s get accessor, which returns the value of courseName.

The first line of the output shows an empty name (marked by single quotes, ''). Unlike local variables, which are not automatically initialized, every field has a default initial value—a value provided by C# when you do not specify the initial value. Thus, fields are not required to be explicitly initialized before they’re used in an app unless they must be initialized to values other than their default values. The default value for an instance variable of type string (like courseName) is null. When you display a string variable that contains the value null, no text is displayed on the screen.

Line 18 prompts the user to enter a course name. Line 19 assigns the course name entered by the user to object myGradeBook’s CourseName property. When a value is assigned to CourseName, the value specified (which is returned by ReadLine in this case) is assigned to implicit parameter **value** of CourseName’s set accessor. Then parameter **value** is assigned by the set accessor to instance variable courseName.

Updated UML class diagram showing how to represent a Property:



Surely this still allows external methods to access the class’s private instance variables, albeit through the property, so this is not protecting the data? The difference is that using the property allows us to control the access to the data, so for the get accessor we can choose not to reveal the raw data and possible return an interpretation of the data i.e. a class could store the number of seconds since midnight, but the get could choose to return the hours, minutes and seconds (typically a get accessor does return the raw data). The set accessor is where we see the main advantage because we can add code to the set accessor to validate the data before we store it in the instance variable.

Class methods should also use the properties to access the instance variable even although they are permitted to access them directly. This makes for a more robust class because if we decide to change the representation of the instance variable, only the get and set accessors that directly manipulate the instance variable need to change.

Auto-implemented Properties

If, as in our example above, you are simply using the get accessor to return the value of the instance variable, and the set accessor to assign a value to the instance variable then we can use the auto-implemented properties.

C# provides automatically implemented properties (also known as auto-implemented properties). With an auto-implemented property, the C# compiler creates a private instance variable, and the get and set accessors for returning and modifying the private instance variable. Unlike a user-defined property, an auto-implemented property, must have both a get and a set accessor. This enables you to implement the property trivially, which is handy when you’re first designing a class. If you later decide to include other logic in the get or set accessors, you can simply modify the property’s implementation. To use an auto-implemented property in the GradeBook class, you can replace the private instance variable at line 11 and the property at lines 14–26 with the following code:

public string CourseName { get; set; }