Encapsulation

The concept of **encapsulation** revolves around the notion that an object's data should not be directly accessible from an object instance.

To be more specific, members of a class that represent an object's state should not be marked as public.

Encapsulation provides a way to preserve the integrity of an object's state data. Rather than defining public fields (which can easily foster data corruption), you should get in the habit of defining private data, which is indirectly manipulated using one of two main techniques.

- You can define a pair of public accessor (get) and mutator (set) methods.
- You can define a public .NET property.

Whichever technique you choose, the point is that a well-encapsulated class should protect its data and hide the details of how it operates from the prying eyes of the outside world.

Encapsulation Using Traditional Accessors and Mutators

```
class Employee
{
        // Field data.
        private string empName;

        // Accessor (get method).
        public string GetName()
        {
                return empName;
        }

        // Mutator (set method).
        public void SetName(string name)
        {
                // Do a check on incoming value before making assignment.
                if (name.Length > 15)
                Console.WriteLine("Error! Name length exceeds 15 characters!");
                else
                empName = name;
        }
}
static void Main(string[] args)
{
        Employee employee1 = new Employee();
        employee1.SetName("Helen");
        employee1.GetName();
}
```

If you want the outside world to interact with a worker's full name, a traditional approach is to define an accessor (get method) and a mutator (set method). The role of a get method is to return to the caller the current value of the underlying state data. A set method allows the caller to change the current value of the underlying state data, as long as the defined business rules are met.

Encapsulation Using .NET Properties

Although you can encapsulate a piece of field data using traditional get and set methods, .NET languages prefer to enforce data encapsulation state data using properties. First, understand that **properties** are just a simplification for "real" accessor and mutator methods.

A C# **property** is composed by defining a get scope (accessor) and set scope (mutator) directly within the property itself. Notice that the property specifies the type of data it is encapsulating by what appears to be a return value. Also take note that, unlike a method, properties do not make use of parentheses (not even empty parentheses) when being defined.

```
public class Employee
{
        private string empName;
        public string Name
        {
                get { return empName; }
                set
                {
                        if (value.Length > 15)
                                Console.WriteLine("Error! Name length exceeds 15
                        characters!");
                        else
                                empName = value;
                }
        }
}
static void Main(string[] args)
{
        Employee employee1 = new Employee();
        employee1.Name = "John";
}
```

In this example we encapsulate empName data field by using Name property,

Within a set scope of a property, you use a token named value, which is used to represent the incoming **value** used to assign the property by the caller. This token is not a true C# keyword but is what is known as a contextual keyword. When the token value is within the set scope of the property, it always represents the value being assigned by the caller, and it will always be the same underlying data type as the property itself.

Properties as Expression-Bodied Members

Property get and set accessors can also be written as expression-bodied members.

```
public int Age
{
        get => empAge;
        set => empAge = value;
}
```

Beyond updating constructors to use properties when assigning values, it is good practice to use properties throughout a class implementation to ensure your business rules are always enforced. In many cases, the only time when you directly make reference to the underlying private piece of data is within the property itself.

Read-Only and Write-Only Properties

When encapsulating data, you might want to configure a read-only property. To do so, simply omit the set block. Likewise, if you want to have a write-only property, omit the get block.

```
public class Person
{
        private string personName;
        private int personAge;

        // Read-only property
        public string Name
        {
                get { return personName; }
        }

        // Write-only property
        public int Age
        {
                set { personAge = value; }
        }
}
```

It is important to know that we can also formalize *static properties* using static keyword.

Authomatic Properties

When you are building properties to encapsulate your data, it is common to find that the set scopes have code to enforce business rules of your program. However, in some cases you may not need any implementation logic beyond simply getting and setting the value. This means you can end up with a lot of code looking like the following:

```
class Car
{
        private string carName = "";
        public string PetName
        {
                get { return carName; }
                set { carName = value; }
        }
}
```

To streamline the process of providing simple encapsulation of field data, you may use **automatic property** syntax. As the name implies, this feature will offload the work of defining a private backing field and the related C# property member to the compiler using a new bit of syntax.

```
private string CarName { get; set; }
```

When defining automatic properties, you simply specify the access modifier, underlying data type, property name, and empty get/set scopes. At compile time, your type will be provided with an autogenerated private backing field and a fitting implementation of the get/set logic.

```
// Read-only property? This is OK!
public int MyReadOnlyProp { get; }

// Write only property? Error!
public int MyWriteOnlyProp { set; }
```

Because the compiler will define the private backing field at compile time (and given that these fields are not directly accessible in C# code), the class-defining automatic properties will always need to use property syntax to get and set the underlying value.

Automatic Properties and Default Values

When you use automatic properties to encapsulate numerical or Boolean data, you are able to use the autogenerated type properties straightaway within your codebase, as the hidden backing fields will be assigned a safe default value (false for Booleans and 0 for numerical data). However, be aware that if you use automatic property syntax to wrap another class variable, the hidden private reference type will also be set to a default value of null (which can prove problematic if you are not careful).

```
class Garage
{
        // The hidden int backing field is set to zero!
        public int NumberOfCars { get; set; }

        // The hidden Car backing field is set to null!
        public Car MyAuto { get; set; }
}

static void Main(string[] args)
{
        Garage g = new Garage();

        // OK, prints default value of zero.
        Console.WriteLine("Number of Cars: {0}", g.NumberOfCars);

        // Runtime error! Backing field is currently null!
        Console.WriteLine(g.MyAuto.PetName);
}
```

In this case we will receive a "null reference exception" at runtime, as the Car member variable used in the background has not been assigned to a new object. To solve this problem, you could update the class constructors to ensure the object comes to life in a safe manner.

```
class Garage
{
    // The hidden int backing field is set to zero!
    public int NumberOfCars { get; set; }
    // The hidden Car backing field is set to null!
    public Car MyAuto { get; set; }
    // Must use constructors to override default
    // values assigned to hidden backing fields.
    public Garage()
    {
        MyAuto = new Car();
        NumberOfCars = 1;
    }
    public Garage(Car car, int number)
    {
        MyAuto = car;
        NumberOfCars = number;
    }
}
```

Initialization of Automatic Properties

A data field of a class can be directly assigned an initial value upon declaration.

```
class Garage
{
    public int NumberOfCars = 10;
}

class Garage
{
    // The hidden backing field is set to 1.
    public int NumberOfCars { get; set; } = 1;
    // The hidden backing field is set to a new Car object.
    public Car MyAuto { get; set; } = new Car();
    public Garage() { }
    public Garage(Car car, int number)
    {
        MyAuto = car;
        NumberOfCars = number;
    }
}
```

In a similar manner, C# allows you to assign an initial value to the underlying backing field generated by the compiler.

Understanding Object Initialization Syntax

C# offers *object initializer syntax* using which it is possible to create a new object variable and assign a slew of properties and/or public fields in a few lines of code. Syntactically, an object initializer consists of a comma-delimited list of specified values, enclosed by the { and } tokens. Each member in the initialization list maps to the name of a public field or public property of the object being initialized.

```
class Point
{
	public int X { get; set; }
	public int Y { get; set; }

	public Point() { }

	public Point(int xVal, int yVal)
	{
		X = xVal;
		Y = yVal;
	}
}

static void Main(string[] args)
{
	// Make a Point by setting each property manually.
	Point firstPoint = new Point();
	firstPoint.X = 10;
	firstPoint.Y = 10;

	// Or make a Point via a custom constructor.
	Point anotherPoint = new Point(20, 20);

	// Or make a Point using object init syntax.
	Point finalPoint = new Point
	{
		X = 30,
		Y = 30
	};
}
```

Calling Custom Constructors with Initialization Syntax

```
// Here, the default constructor is called implicitly.
Point finalPoint = new Point { X = 30, Y = 30 };
```

It is permissible to explicitly call the default constructor as follows:

```
// Here, the default constructor is called explicitly.
Point finalPoint = new Point() { X = 30, Y = 30 };
```

If we are constructing a type using initialization syntax, you are able to invoke any constructor defined by the class.

```
// Calling a custom constructor.
Point finalPoint = new Point(25, 20)
{
        X = 100,
        Y = 150
};
// In this case X point of finalPoint object wiil be 100 and Y point will be 150.
```

Constant Field Data

C# offers the const keyword to define constant data, which can never change after the initial assignment. This can be helpful when you are defining a set of known values for use in your applications that are logically connected to a given class or structure.

The initial value assigned to the constant must be specified at the time you define the constant. The value of constant data must be known at compile time. Constructors (or any other method), as you know, are invoked at runtime.

```
class MyMathClass
{
        public const double PI = 3.14;
}
```

Read-Only Fields

Like a constant, a **read-only field cannot be changed after the initial assignment**. However, unlike a constant, **the value assigned to a read-only field can be determined at runtime and, therefore, can legally be assigned within the scope of a constructor but nowhere else.**

This can be helpful when you don't know the value of a field until runtime, perhaps because you need to read an external file to obtain the value but want to ensure that the value will not change after that point.

```
class MathClass
{
	public static readonly double ENumber = 2.718;
	// or
	public readonly double PI;
	public MathClass()
	{
		PI = 3.14;
	}
}
```

Any attempt to make assignments to a field marked readonly outside the scope of a constructor results in a *compiler error*.

Unlike a constant field, read-only fields are not implicitly static. Thus, if you want to expose PI from the class level, you must explicitly use the static keyword.

However, if the value of a static read-only field is not known until runtime, you must use a static constructor as described earlier in this chapter.

Partial Classes

Using partial classes, you could choose to move (for example) the properties, constructors, and field data into a new file. The first step is to add the partial keyword to the current class definition and cut the code to be placed into the new file. When you have inserted a new class file into your project, you can move the data fields and constructors to the new file using a simple cut-and-paste operation. In addition, you must add the partial keyword to this aspect of the class definition.

After you compile the modified project, you should see no difference whatsoever. The whole idea of a partial class is realized only during design time. *After the application has been compiled, there is just a single, unified class within the assembly. The only requirement when defining partial types is that the type's name is identical and defined within the same .NET namespace.*

```
partial class Country
{
	// Fields
	// Properties
}

partial class Country
{
	// Constructors
	// Methods
}
```