Pillars of OOP

According to the book "Pro C# 7 With .NET and .NET Core" of Andrew Troelsen all object-oriented languages (C#, Java, C++, Visual Basic, etc.) must contend with three core principles, often called the pillars of object-oriented programming (OOP).

- **Encapsulation**: How does this language hide an object's internal implementation details and preserve data integrity?
- **Inheritance**: How does this language promote code reuse?
- **Polymorphism**: How does this language let you treat related objects in a similar way?

Before digging into the syntactic details of each pillar, it is important to understand the basic role of each. Here is an overview of each pillar.

Encapsulation

This trait boils down to the language's ability to hide unnecessary implementation details from the object user. Closely related to the notion of encapsulating programming logic is the idea of data protection. Ideally, an object's state data should be specified using the private (or possibly protected) keyword. In this way, the outside world must ask politely in order to change or obtain the underlying value. This is a good thing, as publicly declared data points can easily become corrupted (ideally by accident rather than intent!).

Inheritance

Inheritance boils down to the language's ability to allow you to build new class definitions based on existing class definitions. In essence, inheritance allows you to extend the behavior of a base (or parent) class by inheriting core functionality into the derived subclass (also called a child class).

Under the .net platform, System.Object is always the topmost parent in any class hierarchy, which defines some general functionality for all types.

Polymorphism

This trait captures a language's ability to treat related objects in a similar manner. Specifically, this tenant of an object-oriented language allows a base class to define a set of members (formally termed the *polymorphic interface*) that are available to all descendants. A class's polymorphic interface is constructed using any number of *virtual* or *abstract* members.

A *virtual member* is a member in a base class that defines a default implementation that may be changed (or more formally speaking, *overridden*) by a derived class. In contrast, an *abstract method* is a member in a base class that does not provide a default implementation but does provide a signature. When a class derives from a base class defining an abstract method, it *must* be overridden by a derived type. In either case, when derived types override the members defined by a base class, they are essentially redefining how they respond to the same request.

Access Modifiers

Types (classes, interfaces, structures, enumerations, and delegates) as well as their members (properties, methods, constructors, and fields) are defined using a specific keyword to control how "visible" the item is to other parts of your application. Although C# defines numerous keywords to control access, they differ on where they can be successfully applied.

- **public** - The type or member can be accessed by any other code in the same assembly or another assembly that references it.
- **private -** The type or member can be accessed only by code in the same class or struct.
- **protected -** The type or member can be accessed only by code in the same class, or in a class that is derived from that class.
- **internal** - The type or member can be accessed by any code in the same assembly, but not from another assembly.
- **protected internal -** The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly.
- **private protected -** The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class.

It is important to know that the *internal* and *protected internal* modifiers are useful when you build .NET code libraries and the protected modifier is useful when you are creating class hierarchies.

***By default**, type members are implicitly private while types are implicitly internal.*

```
// An internal class with a private default constructor.
class Radio
{
        Radio() { }
}
```

As we mentioned the *private, protected*, and *protected internal* access modifiers can be applied to a *nested type*. Nested type is a type declared directly within the scope of class or structure.

```
public class SportsCar
{
        // OK! Nested types can be marked private.
        private enum CarColor
        {
                Red,
                Green,
                Blue
        }
}
```

However, non-nested types can be defined only with the public or internal modifiers.