Note2:C# as OOP

Some of the Strong Programming Features of C# are:

- Boolean Conditions
- Automatic Garbage Collection
- Standard Library
- Assembly Versioning
- Properties and Events
- Delegates and Events Management
- Easy-to-use Generics
- Indexers
- Conditional Compilation
- Simple Multithreading
- LINQ and Lambda Expressions
- Integration with Windows

A C# program consists of the following parts –

- Namespace declaration
- Class
- Class methods
- Class attributes
- Main method
- Statements and Expressions
- Comments

```
//Basic Structure of a C# Program
namespace HelloWorld
{
   class Program
   {
      static void Main(string[] args)
      {

      }
   }
}
```

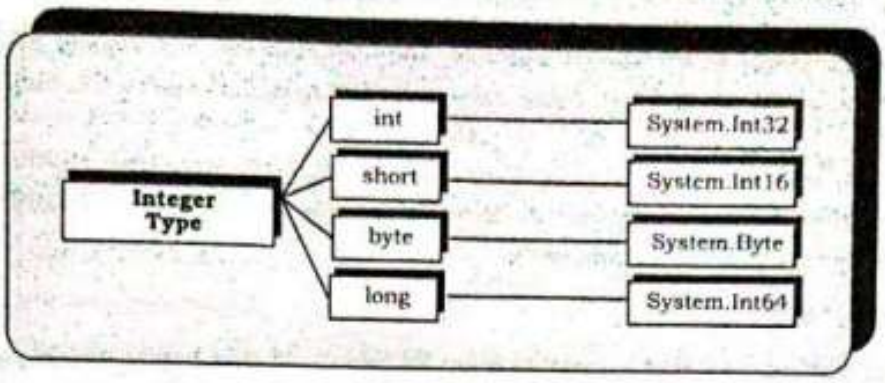

Figure 6: Integer Type Hierarchy

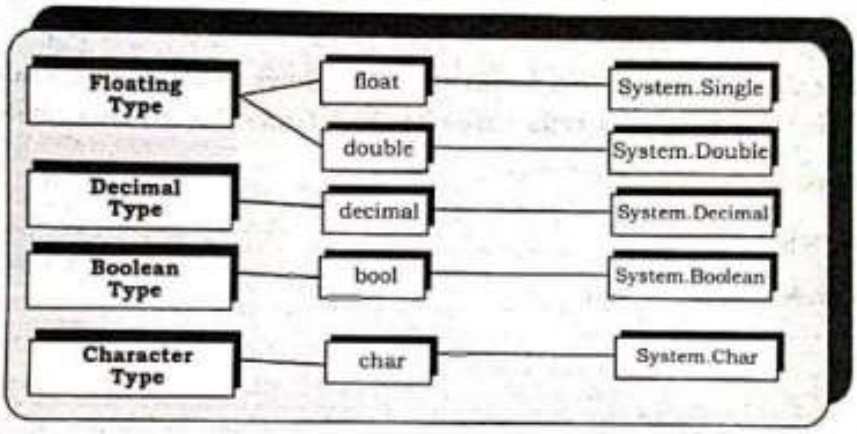

Figure 7: Other Value Type Hierarchy

Reference Type

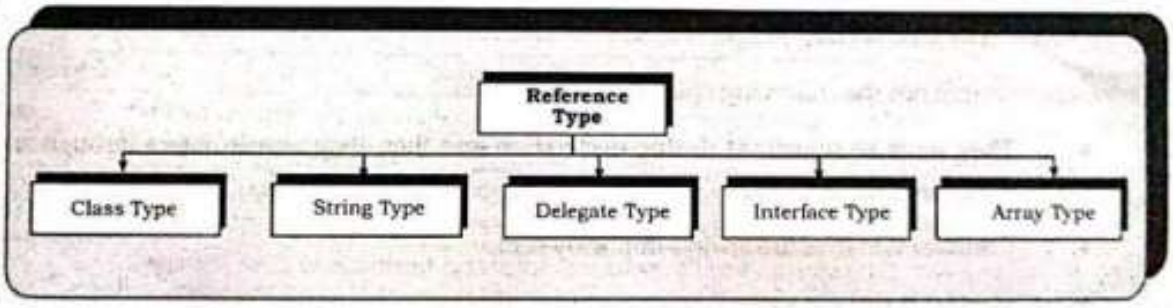

Figure 8: Reference Type Hierarchy

Object-Oriented Programming (c.f. Ch 5: Mark J. Price 2018)

An object in the real world is a thing, such as a car or a person, whereas an object in programming often represents something in the real world, such as a product or bank account, but this can also be something more abstract. In C#, we use the class (mostly) or struct (sometimes) C# keywords to define a type of object.

Features/Concepts of object-oriented programming

- **Encapsulation**: This is the combination of the data (fields) and methods that are related to an object. For example, a BankAccount type might have data, such as Balance and AccountName, as well as actions, such as Deposit and Withdraw. When encapsulating, you often want to control what can access those actions and the data, for example, restricting how the internal state of an object can be accessed or modified from the outside.

- **Composition**: This is about what an object is made of. For example, a car is composed of different parts, such as four wheels, several seats, and an engine.

- **Aggregation**: This is about what can be combined with an object. For example, a person is not part of a car object, but they could sit in the driver's seat and then becomes the car's driver. Two separate objects that are aggregated together to form a new component.

- **Inheritance**: This is about reusing code by having a subclass derive from a base or super class. All functionality in the base class is inherited

by and becomes available in the derived class. For example, the base or super Exception class has some members that have the same implementation across all exceptions, and the sub or derived SqlException class inherits those members and has extra members only relevant to when an SQL database exception occurs like a property for the database connection.

- **Abstraction**: This is about capturing the core idea of an object and ignoring the details or specifics. C# has an abstract keyword which formalizes the concept. If a class is not explicitly abstract then it can be described as being concrete. Base or super classes are often abstract, for example, the super class Stream is abstract and its sub classes like FileStream and MemoryStream are concrete. Abstraction is a tricky balance. If you make a class more abstract, more classes would be able to inherit from it, but at the same time there will be less functionality to share.

- **Polymorphism**: is about allowing a derived class to override an inherited action to provide custom behavior.

Building class libraries

Class library assemblies group types together into easily deployable units (DLL files). To make the code that you write reusable across multiple projects, we should put it in class library assemblies, just like Microsoft does. We will see this in action in the lab.

The first POCO class

```
public class Person : object
{
 // fields
 public string Name;
 public DateTime DateOfBirth;
}
```

After defining the class in a separate .cs class file, we can use/consume it in the main program as below:

```
var bob = new Person(); //Instantiating a class
bob.Name = "Bob Smith";
bob.DateOfBirth = new DateTime(1965, 12, 22);
WriteLine(
 format: "{0} was born on {1:dddd, d MMMM yyyy}",
 arg0: bob.Name,
 arg1: bob.DateOfBirth);
```

Understanding members of the class

Members can be fields, methods, or specialized versions of both.

Fields are used to store data. There are also three specialized categories of field:

° Constant: The data never changes. The compiler literally copies the data into any code that reads it.

° Read-only: The data cannot change after the class is instantiated, but the data can be calculated or loaded from an external source at the time of instantiation.

° Event: The data references one or more methods that you want to execute when something happens, such as clicking on a button, or responding to a request from other code.

Methods are used to execute statements.
There are also four specialized categories of method:

° **Constructor**: The statements execute when you use the new keyword to allocate memory and instantiate a class.

° **Property**: The statements execute when you get or set data. The data is commonly stored in a field, but could be stored externally, or calculated at runtime. Properties are the preferred way to encapsulate fields unless the memory address of the field needs to be exposed.
° **Indexer**: The statements execute when you get or set data using array syntax [].
° **Operator**: The statements execute when you use an operator like + and / on operands of your type

Referencing an assembly
Before we can instantiate a class, we need to reference the assembly that contains it.

Change the .csproj file as below:

```
<Project Sdk="Microsoft.NET.Sdk>
 <PropertyGroup>
 <OutputType>Exe</OutputType>
 <TargetFramework>netcoreapp3.0</TargetFramework>
 </PropertyGroup>
 <ItemGroup>
 <ProjectReference Include="../PacktLibrary/PacktLibrary.csproj" />
```

```
 </ItemGroup>
</Project>
```

Understanding access modifiers

Part of encapsulation is choosing how visible the members are.

Access Modifier	Description
`private`	Member is accessible inside the type only. This is the default.
`internal`	Member is accessible inside the type and any type in the same assembly.
`protected`	Member is accessible inside the type and any type that inherits from the type.
`public`	Member is accessible everywhere.
`internal protected`	Member is accessible inside the type, any type in the same assembly, and any type that inherits from the type. Equivalent to a fictional access modifier named `internal_or_protected`.
`private protected`	Member is accessible inside the type, or any type that inherits from the type and is in the same assembly. Equivalent to a fictional access modifier named `internal_and_protected`. This combination is only available with C# 7.2 or later.

Storing a value using an enum type

Sometimes, a value needs to be one of a limited set of options. For example, there are seven ancient wonders of the world, and a person may have one favorite.

```
namespace Packt.Shared
{
 public enum WondersOfTheAncientWorld
```

```
{
GreatPyramidOfGiza,
HangingGardensOfBabylon,
StatueOfZeusAtOlympia,
TempleOfArtemisAtEphesus,
MausoleumAtHalicarnassus,
ColossusOfRhodes,
LighthouseOfAlexandria
}
}
```

After creating the enum type we can use it in the main program as below:

In the Person class, add the following statement to your list of fields:

```
public WondersOfTheAncientWorld FavoriteAncientWonder;
```

In the Main method of Program.cs, add the following statements:

```
bob.FavoriteAncientWonder =
 WondersOfTheAncientWorld.StatueOfZeusAtOlympia;

WriteLine(format:
 "{0}'s favorite wonder is {1}. It's integer is {2}.",
 arg0: bob.Name,
 arg1: bob.FavoriteAncientWonder,
 arg2: (int)bob.FavoriteAncientWonder);
```

Run the application and view the result, as shown in the following output:

```
Bob Smith's favorite wonder is StatueOfZeusAtOlympia. Its
integer is 2.
```