Summary

* C# provides full support for object-oriented programming including encapsulation, inheritance, and polymorphism.

***Encapsulation*** means that a **group of related** properties, methods, and other members are **treated as a single** unit or object.

***Inheritance*** describes the ability to create **new** classes **based on an existing** class.

***Polymorphism*** means that you can have **multiple classes** that **can be used interchangeably,** even though each class implements the same properties or methods in different ways.

**Classes and Objects**

* classes describe the *type* of objects,
* objects are usable *instances* of classes.
* the act of creating an object is called *instantiation*.

Using the blueprint analogy, a class is a blueprint, and an object is a building made from that blueprint.

* *structures -* a light version of classes - that are useful when you need to create large array of objects and do not want to consume too much memory for that.

**Class Members**

Each class can have different *class members* that include

* **properties** that **describe class data,**
* methods that define class behavior,
* events that provide communication between different classes and objects.

**Properties and Fields**

**Fields** and **properties** represent **information that an object contains**. Fields are like variables because they can be read or set directly.

**Properties** have **get and set procedures**, which provide more control on how values are set or returned.

**Methods**

A *method* is an action that an object can perform.

A class can have several implementations, or *overloads*, of the same method that differ in the number of parameters or parameter types.

**Constructors**

**Constructors** are **class methods** that are executed automatically when an object of a given type is created.

**Constructors usually initialize the data members of the new object.**

**Nested Classes**

A class defined within another class is called *nested*. By default, the nested class is private.

**Access Modifiers and Access Levels**

* **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 only be accessed by code in the same class
* **protected** - the type or member can only be accessed by code in the same class or in a derived 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 same assembly, or by any derived class in another assembly
* **private protected -** the type or member can be accessed by code in the same class or in a derived class within the base class assembly

**Static Classes and Members**

A static member of the class is a property, procedure, or field that **is shared by all instances of a class.**

**Static classes** in C# have **static members only** and **cannot be instantiated**. Static members also cannot access non-static properties, fields or methods

**Overriding Members**

* By default, a derived class inherits all members from its base class.
* If you want to change the behavior of the inherited member, you need to override it. That is, you can define a new implementation of the method, property or event in the derived class.

The following modifiers are used to control how properties and methods are overridden:

* **virtual** - Allows a class member to be overridden in a derived class
* **override** - Overrides a virtual (overridable) member defined in the base class
* **abstract** - Requires that a class member to be overridden in the derived class
* **new Modifier** - Hides a member inherited from a base class

**Interfaces**

Interfaces define a set of properties, methods, and events. But **unlike classes**, **interfaces do not provide implementation**.

* They are **implemented by classes**, **and defined as separate entities** from classes.
* An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.