Property:

The property is the extension on the variables that provides the way to configure the access specification of the data member or variable. It can also act as the way to declaration of the variable and configure its access specification.

Syntax of simple property and variable declaration:

```
private Hours;

public double Hours { get; set; }
```

In the above example:

- There exists a private variable of double type.
- The public property of the type double which will be used to define the get and set accessor on the private variable.

There are two types of accessors in the property:

1. get: The get accessor/method in the property is used to define the read-access on the property. If the property only consists of the get accessor, then it said to be the read-only property.
   For Example:-
   ```
   private int mfgDate;
   public int Date { get; }
   ```
2. set: The set accessor/method in the property is used to define the write-access on the property. If the property only consists of the set accessor, then it said to be the write-only property. In this some additional logic maybe regarding validation and verification can be used to customize the write-only functionality of the variable.
   For Example:-
   ```
   private int mfgYear;
    public int Year { set{
           if(value>10)
                  mfgYear = value;
           else
                  mfgYear = 10;
   } }
   ```

In the above example, the value given will assigned to the variable only when the value provided is greater than 10, otherwise the value as 10 will be stored. The value is keyword is use to access the provided value that needs to be set in the variable.

Both of the accessors can be used with-in the property and provides the write and read access of the method.

```
private int mfgYear;
     public int Year { set{
            if(value>10) mfgYear = value;
```

```
                else mfgYear = 10;
            }
        get;  }
```

Access-modifiers on the properties:

In the above example, the public property (Year) is created which use the private variable (mfgYear of type: int). The access of the property can also be modified using:

- Public
- Private
- Protected
- Internal

Different types of property implementation:

1. **Auto-implemented implementation:** In this implementation, both the set and get accessors are used within the property without any additional logic.
   Example:
   ```
   public float percentage{
   get; set; }
   ```
2. **Expression:** In this, if the accessors only consist of single line logic then the **=>** can be used to define the single line expression of the accessor. If the multiline logic is used then the block of the accessors need to created using the **{}**.
   Single Line Example:
   ```
     private float percent;
   public float percentage{
             get => percent;
                set = > percent = value; }
   ```
   Block Example:
   ```
   private int mfgYear;
   public int Year { set{
           if(value>10) mfgYear = value;
            else mfgYear = 10;
   }
   get{
           return mfgYear;}  }
   ```

Constructor

The constructor is the member of the class which will execute itself on the creation of the object of that particular class. It is may be used to initialize the data-members of the class or it may be used to do some logic on the creation of the class.

The C# by default implicitly consists of a constructor which is known as Default Constructor. The constructor can also be overloaded and thus can be **categorized** into:

1. Parameter-less Constructor: are those in which no parameters are passed. Default constructor is also the of this type.
   Syntax: `public <class-name>(){ }`
   Example:
   ```
   class System{
           private int RAMSize;
   private int ROMSize;
           public System(){
           RAMSize = 4;
           ROMSize = 500;
           }}
   ```
2. Parameterized Constructor: are those in which the parameters are passed according to different logic.
   Syntax: `public <class-name> (parameter-list){ }`
   Example:
   ```
   class System{
           private int RAMSize;
           private int ROMSize;
           public System(int ramSize, int romSize){
           RAMSize = ramsize;
           ROMSize = romsize;
           }}
   ```
3. Copy Constructor: are used to copy the data from one object to another new object. In this, the object of the class is passed as an argument.
   Syntax: `public <class-name>(<class-name> <obj-name>){}`
   Example:
   ```
           class System{
           private int RAMSize;
           private int ROMSize;
           public System(int ramSize, int romSize){
           RAMSize = ramsize;
           ROMSize = romsize;
           }
   public System(System sys){
           this.RAMSize = sys.RAMSize;
           this.ROMSize = sys.ROMSize;
           }}
   ```

4. Static Constructor: is a different type of constructor which is called automatically before any object of the class is created. As it's a static block then only it will deals with only static data-members and methods of the class. It is a parameter-less constructor which does not take any parameters.

   Syntax: `public <class-name>(){}`

   Example:

```
        class System{
        static string cabinet;
        private int RAMSize;
        private int ROMSize;
static System(){
        cabinet = "ATX";}
        public System(int ramSize, int romSize){
        RAMSize = ramsize;
        ROMSize = romsize;
        }
public System(System sys){
        this.RAMSize = sys.RAMSize;
        this.ROMSize = sys.ROMSize;
        }}
```

5. Private Constructor: is the one which cannot be accessed outside the class, hence if a class consists of a only the single constructor then the object of the class cannot be created outside the class. The object of the class can be created within the members of the class such as functions or nested classes. It can be parameterized or parameter-less.

   Syntax: `private <class-name>(){}`

   Example:

```
                class System{
        static string cabinet;
        private int RAMSize;
        private int ROMSize;
private System(){
        cabinet = "ATX";}}
```

Virtual:

It is a keyword which is used in the parent class so that the function can be overridden by the derived class or the child class.

Syntax: <access-modifier> virtual <return-type> <function-name>(parameter-list){}

Example:

```
class Vehicle{
        public int milage { get; set; }
        public string type { get; set; }
        public virtual int Engine(){
        Console.WriteLine("Parent Engine v8"); }
        }
```

Override:

It is a keyword which is used in the child class so that the function can be override by the class. If the child class did not override the function then the function of the parent class can be used by the base class.

Syntax: <access-modifier> virtual <return-type> <function-name>(parameter-list){}

Example:

```
class Car: Vehicle{
        public override int Engine(){
        Console.WriteLine("Child Engine v12"); }
        }
```

New:

It is used in the child class don't want to override the function of the parent class, hence hides the function of the parent class.

Syntax: <access-modifier> virtual <return-type> <function-name>(parameter-list){}

Example:

```
class Car: Vehicle{
        public new int Engine(){
        Console.WriteLine("Child Engine v12"); }
        }
```