# **OOP: ـــ**

* **The Pros of OOP:**
* Reusability.
* DRY (Don't Repeat Yourself).
* Clear Structure, so save Debugging and developing.
* **Examples :**

|  |  |
| --- | --- |
| **Class** | **Objects** |
| Car | Skoda, BMW, Audi. |
| Animal | Tiger, Lion, Elephant. |

# **Classes: ـــ**

* **It is a good practice to start with an uppercase first letter when naming classes.**

# **Create Object: ـــ**

* **To Create an object from a class :**

**Car** myObj1 = new Car();

**var** myObj2 = new Car();

* **A static method can be accessed without creating an object of the class, while public methods can only be accessed by objects.**

# **Constructor: ـــ**

* **Constructor is used to initializing objects.**
* Constructor is called **when** the object is created.
* Constructor's name must **match** the **class name** and **cannot** have a **return type**.
* **All** classes have constructors **by default**.
* Constructors can be **overloaded** by using **different numbers of parameters**.
* **Types of Constructors:**

1. **Default Constructor**:

* The default constructor initializes all **numeric fields** to **zero** and **all string and object fields** to **null**.

1. **Parameterized Constructor**:

* A constructor having **at least one parameter**.

1. **Copy Constructor**:

* Its main use is to initialize **a new instance** **to the values of** **an** **existing instance**

1. **Private Constructor**:

* It is **not possible** for **other classes to derive from this class** and also it’s **not possible** to **create an instance of this class**.
* It is the implementation of a **singleton class pattern**.
* use a private constructor **when we have only static members.**

1. **Static Constructor**:

* Static Constructor has to be **invoked only once in the class** and it has been **invoked during the creation of the first reference to a static member** in the class.
* A static constructor is **initialized static fields or data of the class and to be executed only once**.
* It **can’t be** called directly.
* When it is executing then the user has **no control**.
* It **does not take** **access modifiers** or any **parameters**.
* It is **called automatically** to initialize the class before the first instance is created.

# **Encapsulation: ـــ**

* As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.
* **Encapsulation can be achieved by:** Declaring all the variables in the class as private and using [C# Properties](https://www.geeksforgeeks.org/c-properties/) in the class to set and get the values of variables.
* **Data-Binding:**

# **Access Modifiers: ـــ**

1. **Public:**

* accessible **for all classes** (in the same assembly or another assembly that references it).

1. **Private:**

* It can **only** be accessed **within the same class.**
* **Use a Private modifier** **to achieve "Encapsulation"** - which is the process of making sure that "sensitive" data is hidden from users.
* Provide **public get and set methods**, through **properties**, **to access** and **update** the value of a private field.
* By default, all members of a class are private if you don't specify an access modifier.

1. **Protected:**

* Can be accessed **only** by code **in the same class**, **or** in a class that is derived from that class(**subclass**)(**Inheritance**).

1. **Internal:**

* Can be accessed by any code **in the same assembly**, **but not** **from another assembly**.

1. **Protected Internal:**

* Can be accessed by any code **in the assembly in which it's declared**, **or** **from within a derived class in another assembly**.

1. **Private Protected:**

* Can be **accessed by types derived from the class that are declared within its containing assembly.**

# **Properties: ـــ**

private string name; // field

public string Name // property

{

get { return name; }

set { name = value; }

}

**The Same:**

public string Name { get; set; } // property

* Fields can be made **read-only** (if you only use the get method), or **write-only** (if you only use the set method).

# **Inheritance: ـــ**

* **To inherit from a class, use the : symbol.**

class Car : Vehicle

{

...

}

* If you **don't want other classes to inherit from a class**, use the sealed keyword.

sealed class Vehicle

{

...

}

# **Polymorphism: ـــ**

* It occurs when we have many **classes that are related to each other by inheritance**.
* C# **provides an option to override the base class method**, by adding the **virtual** keyword to the method **inside the base class**, and by using the **override** keyword **for each derived class methods**.

# **Abstraction: ـــ**

* **Abstraction** can be achieved with either **abstract classes** or [**interfaces**](https://www.w3schools.com/cs/cs_interface.asp).
* **Abstract class**: **Cannot be used to create objects** (**to access it, it must be inherited** from another class).
* An abstract class can have **both** **abstract** and **regular** **methods**.
* **Abstract method**: Can **only be used in an abstract class**, and it **does not have a body**.
* You are **not allowed** to declare an abstract class as [**Sealed Class**](https://www.geeksforgeeks.org/c-sealed-class/).

# **Interface: ـــ**

* Is a completely "**abstract class**", which can only contain abstract methods and properties.
* By default, members of an interface are **abstract** and **public**.
* You **do not have to use** the override keyword when implementing an interface.
* On implementation of an interface, you must override all of its methods.

# **Exceptions: ـــ**

* Try and Catch:
* The **try** statement allows you **to define a block of code to be tested for errors while it is being executed**.
* The **catch** statement allows you **to define a block of code to be executed if an error occurs in the try block**.
* Finally:
* The finally statement lets you execute code, after try...catch, **regardless** of the result.
* Throw:
* allows you to create a **custom error**.
* The throw statement is used together with an **exception class**.