Chapter Three

Class and Object

Objectives

- **By the end of this chapter, you should be able to:**
  - understand the concepts of classes and objects.
  - create instances of classes.
  - differentiate between instance variables and local variables.
  - understand different types of constructors and their roles.
  - use this keyword to solve shadowing and when chaining constructors.
  - use static fields, methods, classes, constructors, and properties
  - understand when to use read-only and const.
  - differentiate access modifiers in C#.
  - explore the concept of encapsulation in bundling data and methods within a class.
  - explore the use of properties for encapsulating fields with accessors and mutators.
  - use automatic properties.
  - use object initializers for concise object creation.

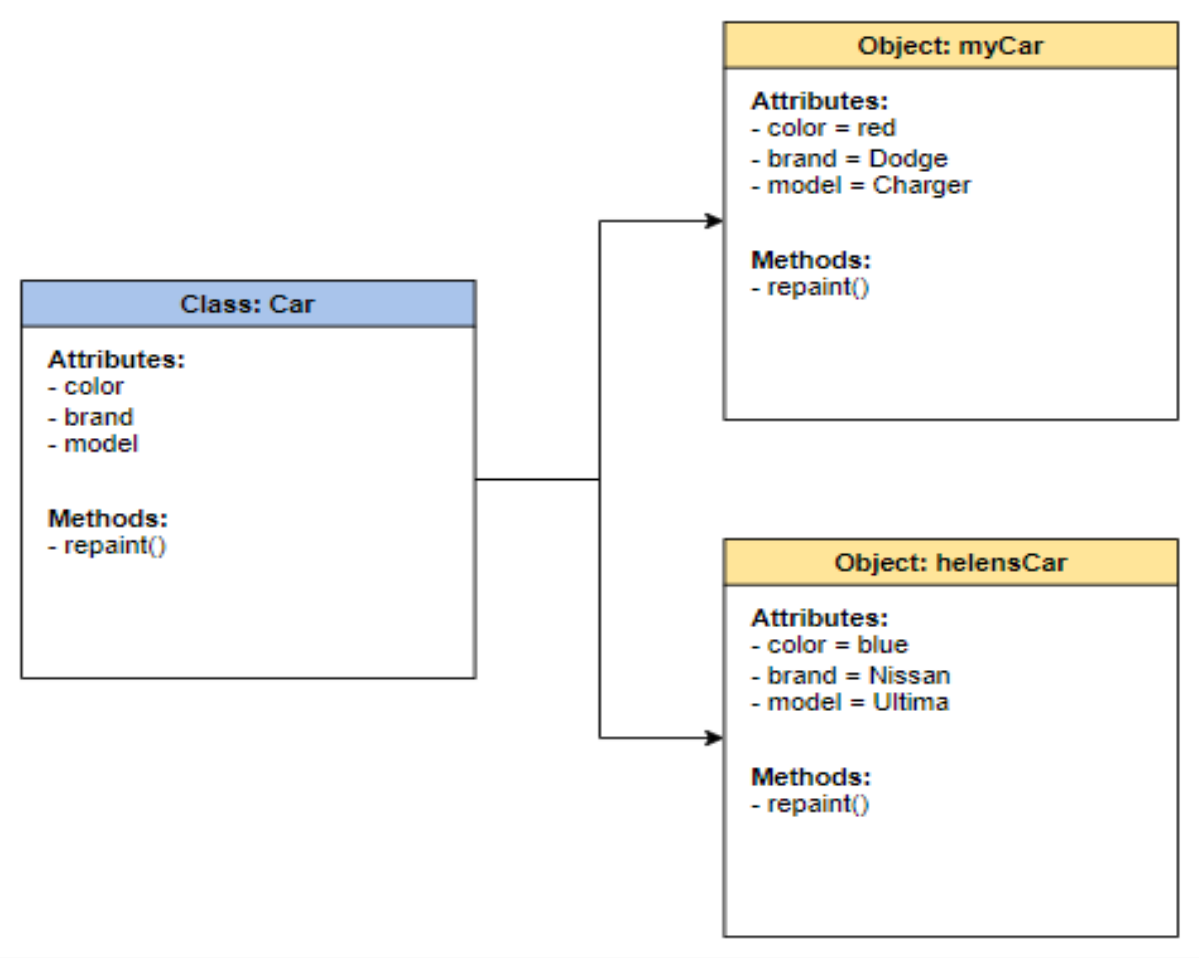
What is Object-Oriented Programming?

- OOP is a programming paradigm/approach that considers everything as an **object**/entity that are instance/example of a particular **class**.
- Object-oriented programming is centered around objects that encapsulate both data and the methods that operate on them.
- **Classes**: are user-defined data types that act as the **blueprint** for individual object's attributes and methods.
- It is a design/format from which real and concrete objects/entities can be created. It often represents broad categories like car, person, animal, student…
- **Objects**: are **instances** of a class created with specifically defined data.

Cont.

- The process of creating an object/instance from a class is called **instantiation**.
- Every object has an attribute/property and behavior/action.
- **Attribute**: A characteristic/property/state of an instance that we are interested in. It is specified by **field data/member data/member variables/instance variables**.
- **Behavior**: an operation/action that we assume an instance can perform on itself. It is specified by **member methods**.
- Note: Classes are reference types.
- Note: You can use Visual Studio to draw class diagrams that map to an actual code.

Example : A Simple Analogy



- Suppose you want to drive a car and make it go faster by pressing its accelerator pedal.
- Before you can drive a car, it has to be designed as a drawing - blue print of the car. - **Class**.
- The drawing includes the **state** of the car like its color, logo … and it also includes the design for how the accelerator pedal works, how the steering wheel works ...etc. - **methods**.
- Before you can drive a car, it must be built from the drawings/blueprint that describes it. - **Instantiation**
- Multiple actual cars can be created from the blueprint.

Why use OOP?

- Modularity and Reusability
- Code Organization
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction
- Ease of Maintenance
- Scalability
- Better Modeling of Real-World Concepts
- Support for Parallel Development

Defining a Class

- A class is defined in C# using the **class** keyword.

Syntax

```
class ClassName
{
    //list member data

    //list  methods
}
```

```
class Person
{
    //member data
    public string fullName;
    public int age;

    //methods
    public int BirthDay()
        =>age++;

    public void PrintState()
    => Console.WriteLine($"Your name is {fullName} " +
            $"and you're {age} years old.");
}
```

Object Instantiation

- Objects must be allocated into memory using the **new** keyword.
- The **object itself** is stored in a garbage-collected **heap** and **its reference is** on **a stack.**

Syntax

```
ClassName objectName = new ClassName();
//or
ClassName objectName2;
objectName2 = new ClassName();
```

- To access members of an object you have to use the dot operator(.)

```
Person person1 = new Person();
person1.fullName = "Abebe Kebede";
person1.age = 26;

person1.BirthDay();
person1.PrintState();
```

Instance variables vs. Local variables

	Instance variables	Local variables
Declaration	Declared within a class but outside of any method. Associated with an instance of the class.	Declared within a method, constructor, or block of code.
Scope	Depending on their access modifiers, they can be accessed within the class as well as other classes through an object.	Limited to the method or block in which they are declared.
Lifetime	Created when the object is instantiated and is eligible for GC when the object is no longer referenced.	Limited to the duration of the method or block in which it is declared.
Location	Stored on the heap memory.	Stored on stack memory.
Initialization	Automatically initialized to their default values.	Must be explicitly initialized before they are used.

Constructors

- A constructor is a special method that is automatically called when an object of a class is created using the new keyword.
- Constructors have the same name as the class and do not have a return type, not even void.
- Constructors can be used for:
  - **Initialization**
  - **Setting Default Values**
  - **Resource Allocation**: Constructors can allocate resources, such as opening files, establishing database connections, etc.
  - **Object Setup**: can perform any necessary setup operations to prepare the object for use.

Types of Constructors

- **Default Constructor:** A parameter-less constructor provided by C# implicitly, only If you **don't** define any constructor for a class. Otherwise, it does not exist.
- **Parameter-less Constructor:** A constructor, that is explicitly defined in a class, with no parameters.
- **Parameterized Constructor:** Takes one or more parameters.
- **Copy Constructor:** Used for creating new objects as copies of existing objects without affecting the original.
- **Private Constructor:** used to prevent the instantiation of a class outside the class. Can be used in classes that provide only static members like **Math**.
- **Static Constructor** – we will discuss this soon.

Examples

```
//Parameterless Constructor
public Person()
{
    fullName = "Abebe";
    age = 18;
}
```

```
//Parameterized Constructor
public Person(string fn, int a)
{
    fullName = fn;
    age=a;
}
```

```
//Copy Constructor
public Person(Person p)
{
    fullName=p.fullName;
    age = p.age;
}
```

```
//private Constructor
private Person() { }
```

Cont.

- You can also use expression-bodied constructors if there is only one statement. (Starting from C# 7.0)

```
public Person(string name)
    => fullName=name;
```

- You can use out parameters with constructors.(Starting from C# 7.3)

```
public Person(string fn, int a,
                out bool legalAge){
    fullName = fn;
    age=a;
    if (age < 18){  legalAge = false; }
    else{  legalAge=true;          }
}
```

Destructor & Garbage Collection(GC)

- When an object is no longer referenced it becomes **eligible** for GC.
- C# provides a **destructor/finalizer** that is **automatically** called(**cannot** be called by the programmer) by the **garbage collector** to **de-allocate** an object from heap memory.
- It **doesn't** take any parameter, it is preceded by a tilde **(~)**, has **no access modifier** and there can **only** be **one** finalizer.
- You can use **finalizers** or the **Dispose** method to free unmanaged resources: **windows, files, network,** and **database connections**.
- The garbage collector and the Dispose method are generally sufficient for managing resources in a managed environment.
- Read more on GC on the C# documentation, & Pro C# 10- chapter 9.

this keyword

- this keyword is a reference to the current instance of the class/struct.
- **Non-static** methods implicitly use **this** keyword to refer to the object's instance variables and other non-static class members.
- Purpose:
  - **Disambiguating/Solving shadowing**(when a local variable with the same name as an instance variable hides the instance variable).
  - **Passing the Current Instance to another class explicitly**
  - **Constructor chaining**: helpful when you have a class that defines multiple constructors with repeated operations.
    - You can achieve the same benefits using **optional** and **named** parameters in a single constructor. How?
- **Note**: You **can not** use this keyword inside a static method. Why?

Examples

Disambiguating

```
public Person(string name, int age)
{
    this.name = name;
    this.age = age;
}
```

Constructor chaining

```
public Person(string name, int age)
{
    this.name = name;
    this.age = age;
}
public Person(int age) : this("", age)
{ /* other statements*/  }
public Person(string name) : this(name,0)
{ /* other statements*/  }
```

Passing the Current Instance using this

```
internal class ex_this
{
    public void Print(Example eg1)
        => Console.WriteLine(eg1.age);
}
class Example
{
    public string name;
    public int age;
    public Example(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
    public void callPrint()
    {
        ex_this e1 = new ex_this();
        e1.Print(this); //Passing the Current Instance to Print()
    }
}
```

```
//inside main()
Example m2 = new Example("Biruk", 23);
m2.callPrint();
```

Static keyword

- It is used to declare members (fields, methods, properties, events, and nested types) that belong to the **type itself**, rather than to instances of the type.
- Static members must be **invoked** directly **from** the **class** level, rather than from an object reference variable.
- It can be used to create:
  - **Shared Resources/State:** for fields or properties that are shared among all instances of a class.
  - **Utility classes:** a class that does not maintain any object-level state and is **not** created with the new keyword. Rather, a utility class exposes all functionality as static members.
  - For **importing static** members with the **using** keyword.
    - using static System.Console;

Uses of Static keyword

- **Static Fields:** shared among all instances of a class, only one copy is created. Persists throughout the lifetime of the application.
- Should be assigned during declaration or in a static constructor.
- **Static method:** can be called on the type itself, without creating an instance of the class. It Can be used to manipulate static data. It **cannot** have **non-static/instance** members in it.
- **Static class:** A static class can **only contain static** members, and it cannot be instantiated. It is often used as a container for utility methods. Static classes **cannot** be **inherited**.
- **Static properties:** we will discuss this soon.
- **Static Constructors:** used to initialize static data of a class before any instance of the class is created or any static member is accessed.

Cont.

- Used to initialize the values of static data when the value is not known at compile time.
- **Does not** have any **access modifier** and **cannot** take any **parameters**. Hence there can **only** be **one** static constructor.
- Only executes one time, regardless of how many objects of the type are created.
- Executes **before** any **instance-level** constructors.
- The runtime invokes the static constructor when it creates an instance of the class or before accessing the first static member invoked by the caller.
- The user has no control over when the static constructor is executed in the program.

Examples

Static Field

```
    static void Main(string[] args)
    {
        MyClass obj1 = new MyClass();
        //Console.WriteLine(obj1.pi);  //Error
        Console.WriteLine(MyClass.pi);
        MyClass.pi = 3.5;
        Console.WriteLine(MyClass.pi);
    }
}
class MyClass
{
    //static field/data
    public static double pi=3.14;
```

Static method & Class

```
    static void Main(string[] args) {
        MyClass.PrintPi(10);
        //MyClass2 obj = new MyClass2(); //Error
        MyClass2.Print();
    }
}
class MyClass {
    //static field/data
    public static double pi=3.14;
    //instance variable
    public int count;
    //static method
    public static void PrintPi(int x) {
        //Console.WriteLine(count); //error
        Console.WriteLine(x); //can have local variables
        Console.WriteLine(pi);
    }
}
//static Class
static class MyClass2 {
    //int a; //intance members are not allowed
    static int x;
    static MyClass2() => x = 10;
    public static void Print() =>Console.WriteLine(x);
}
```

Cont.

Using instance ctor for static data

```
    static void Main(string[] args)
    {
        MyClass myClass1 = new MyClass();
        MyClass.pi = 3.2;
        Console.WriteLine(MyClass.pi);
        MyClass myClass2 = new MyClass();
        Console.WriteLine(MyClass.pi);
    }
}
class MyClass
{
    //static field/data
    public static double pi;
    //instance variable
    public int count;
    public MyClass()
    {
        pi = 3.14; //resets the value
                   //every time an obj is created
        count = 10;
    }
```

Using Static ctor for static data

```
    static void Main(string[] args)
    {
        MyClass myClass1 = new MyClass();
        MyClass.pi = 3.2;
        Console.WriteLine(MyClass.pi);
        MyClass myClass2 = new MyClass();
        Console.WriteLine(MyClass.pi);
    }
}
class MyClass
{
    //static field/data
    public static double pi;
    //instance variable
    public int count;
    //static ctor is only called once
    static MyClass()
    {
        //count  = 10; //error
        pi = 3.14;
    }
```

const & readonly keywords

- **const**
  - Used to declare a **compile-time constant field** or **local constant**, so it must be initialized at the time of declaration.
  - **Constant fields are** implicitly **static.**(How do you access it?)
  - **Only built-in** types may be declared as const.
  - Can it be passed by reference? Why?
- **readonly**
  - Used to declare a **run-time constant field**, so it can be assigned at the time of **declaration** or in the **constructor** of the containing class **only**.
  - It **can** be applied to both **instance** variables and **static** variables, but **not local** variables. (It is **not implicitly static**).
  - To **expose** read-only fields from the **class level**, you must **explicitly** use the **static** keyword. Then use a **static constructor** to assign a value to it.

Example

const

```
{
    static void Main(string[] args)
    {
        Circle c=new Circle();
        Console.WriteLine(c.Area(5)) ;
        //Console.WriteLine(c.pi); //error can't be accessed
        // through an object
        Console.WriteLine(Circle.pi); //correct
    }
}
class Circle
{
    //public const double pi;   //error, must be Initialized
    //pi = 3.1415;
    public const double pi = 3.1415; //constant field, implicitly static

    public double Area(double radius) {
        //pi=3.2; //error, can't be changed

        //const int[] array1 = { 1,2,3}; //can't be applied to arrays and
                                         //user defined types
        const double x = 10; //local constant
        Console.WriteLine(x);
        return pi*radius*radius;
    }
```

readonly

```
    static void Main(string[] args)
    {
        //readonly double interestRate; //error can't be local constant
        double rate=double.Parse(Console.ReadLine());
        Account a1=new Account(rate);
        Console.WriteLine(a1.interestRate); //it is accesses through an obj.
        Console.WriteLine(Account.bankname);
    }
}
class  Account
{
    //readonly double interestRate=0.05; // instance variable-constant

    //or through a ctor
    public readonly int[] array1 = {1,2,3};
    public readonly static string bankname; //static variable-constant
    public readonly double interestRate; // instance variable-constant
    public Account(double rate) => interestRate = rate ;
    static Account()
    {
        bankname = "CBE";
    }
    //void change() => array1 = new int[] { 1, 2 }; //error redaonly can't be
                                                    // assigned inside a method
}
```

Access modifiers

- All types(classes, interfaces, structures, enumerations, and delegates) and type members (properties, methods, constructors, and fields) have an accessibility level.
- The accessibility level controls whether they can be used from other code in your assembly or other assemblies.
- By default, **type members** are implicitly **private**, while **types** are implicitly **internal**.
- **Interface** members are **public** by **default** because the purpose of an interface is to enable other types to access a class or struct.
- A type defined within a class, struct, or interface is called a **nested type**.
- A **structure** can only have **public**, **private,** or **internal** modifiers since it **doesn't** support inheritance.

Cont.

- **public**: The type or type member is accessible from any other assembly or code that references the assembly.
- **private**: The nested type or type member is accessible only within the containing class or struct.
- **protected**: The nested type or type member is accessible within its own class and by derived classes.
- **internal**: The type or type member is accessible only within the same assembly.
- **protected internal**: The nested type or type member is accessible within its own assembly and by derived classes inside or outside of the defining assembly.
- **private protected(**C# 7.2): The nested type or type member is only accessible within the same assembly by derived classes.

Summary of Access Modifiers

Caller's location	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Within the class	✔	✔	✔	✔	✔	✔
Derived class (same assembly)	✔	✔	✔	✔	✔	✘
Non-derived class (same assembly)	✔	✔	✘	✔	✘	✘
Derived class (different assembly)	✔	✔	✔	✘	✘	✘
Non-derived class (different assembly)	✔	✘	✘	✘	✘	✘

Pillars of OOP

- The idea of OOP is dependent on the following principles.
- **Encapsulation** means wrapping up data and methods together into a single unit like a class. It is also built on the idea of data hiding.
- **Inheritance** is the ability to create a new class (child/derived class) from an existing one (parent/base class).
- What is the difference between "**is-a**" vs. "**has-a**" relationship? Which one is **inheritance** and **aggregation**?
- **Polymorphism** is the ability of an object to take on multiple forms.
- It allows the creation of multiple methods with the same name, but with different implementations.
- **Abstraction** means to only show the necessary details to the user.
- It focuses on the essential characteristics of an object while ignoring irrelevant details.

Encapsulation

- An object's data should not be directly accessible from an object instance, because it can lead to data corruption.
- Rather, class data is defined as **private**. To **alter** the state of an object **indirectly**, you can use either of the following techniques:
  - a pair of public **accessor/getter** and **mutator/setter** methods.
  - a public **property.**
- **Accessor/Getter**: responsible for **retrieving** the value of a private field from a class. **Doesn't** have any **parameters**.
- **Mutator/Setter**: responsible for **modifying** the value of a private field within a class. Usually has a **void return type**.
- Every private field that needs to be accessed and modified outside the class needs a pair of setter and getter.

Example

```
    static void Main(string[] args)
    {
        Person p1 = new Person();
        //p1.age = 150; //Error! Cannot directly access private members
        p1.SetName("Abebe");
        p1.SetAge(130); //Logical Error! Age must be b/n 0-125!
        Console.WriteLine(p1.GetName()+"\n"+p1.GetAge());
    }
}
class Person
{
    private string name;
    private int age;
    public Person()
    {
        name= string.Empty;
        age=0;
    }
    public Person(string name, int age)
    {
        SetName(name);
        SetAge(age);
    }
    public string GetName() => name; //getter for name
    public int GetAge() => age; //getter for age
    //setter for name
    public void SetName(string n) {
        if (n.Length > 20)
        {
            Console.WriteLine("Error! Name length exceeds 15 characters!");
        }
        else
        {
            name = n;
        }
    }
    //setter for age
    public void SetAge(int a)
    {
        if (a > 125 || a < 0)
        {
            Console.WriteLine("Error! Age must be b/n 0-125!");
        }
        else
        {
            age = a;
        }
    }
```

Properties

- Properties are just a **container** for "real" accessor and mutator methods, named **get** and **set**, respectively.
- Properties provide a cleaner syntax.
- Properties do not make use of parentheses(not even empty parentheses) when being defined.
- **Within** a **set scope** of a property, you use a token named **value**,
- It represents the value being assigned by the caller, and it will always be the **same underlying** data **type** as the **property** itself
- (New 7.0)Single line properties can be written as expression-bodied members.
- You should use properties inside the class to **avoid** code **duplication** & to **ensure** your business **rules** are always **enforced**.

Example

```
    static void Main(string[] args){
        Person p1 = new Person();
        //p1.age = 150; //Error! Cannot directly access private members
        p1.Name="Abebe"; //this is using the public set accessor
                         //inside the property Name
        p1.Age=130; //this is using the public set accessor
                    //inside the property Age
        WriteLine(p1.Name+"\n"+p1.Age);// this is using the public get accessors
                                       // inside the respective  property
    }
}
class Person {
    private string name;
    private int age;
    private string gender;
    //property for gender, it doesn't do any validation
    public string Gender { get=>gender; set=>gender=value; }
    //property for age
    public int Age {//notice that there is no parentheses
        //get accessor
        get{return age;}
        //set accessor, notice the use value
        set{ //here value is int
            if (value > 125 || value < 0){
                Console.WriteLine("Error! Age must be b/n 0-125!");
```

Cont.

```
        //set accessor, notice the use value
        set{ //here value is int
            if (value > 125 || value < 0){
                Console.WriteLine("Error! Age must be b/n 0-125!");
            } else{ age = value; } } }

    //property for name
    public string Name {
        get => name; //using expression-bodied member
        set{//here value is string
            if (value.Length > 20){
                Console.WriteLine("Error! Name length exceeds 15 characters!");
            } else{name = value;} } }
    public Person() //it uses the property to assign values
    {
        Name= string.Empty;
        Age=0;
        Gender=string.Empty;
    }
    public Person(string name, int age,string gender="None")
    {
        Name=name;
        Age=age;
        Gender = gender;
    }
```

More on Properties

- **Read-Only Properties**: omit the set block.
- You can simplify read-only properties further by using expression body members.
- To assign value to it, use the underlying private member within the constructor.
- **Write-Only Properties**: omit the get block.
- You can **mix Private** and **Public Get/Set** accessors on Properties, to do so declare the **get** method as **public** but the **set** method as **private**.
- The write/set is hidden from anything outside the defining class.
- **Static Properties**: can be used to wrap static fields.
- Static property can access only other static members of the class.

Example

```
class AccountInfo
{
    private int creditcardno;
    private string ssn;
    private int idno;
    private static double interestRate = 0.04;
    public int CreditCardNo => creditcardno; //read-only property
    public int IdNo {  set=> idno=value; }  //write-only property

    public string SocialSecurityNumber //mixing Private and Public Get/Set
    {
        get => ssn;
        private set => ssn = value;
    }
    public AccountInfo(string ssn,int number)
    {//using the private memb. through the ctor
        SocialSecurityNumber = ssn;
        creditcardno=number;
    }
    public static double InterestRate //static prop.
    {
        //get=>idno; //cannot access non-static field from static member
        get=> interestRate;
        set=> interestRate = value;
    }
```

Automatic Properties

- If there is **no need** for additional **logic** when getting and setting a value, you can use **automatic properties**.
- Reduces boilerplate code associated with normal properties.
- Simply specify the access modifier, underlying data type, property name, and empty get/set scopes.
- Behind the scenes, the compiler generates a **private backing field** for the property.
- The compiler-generated backing field is **implicitly named** and **not** directly **accessible** in your code.
- (C# 6)It is possible to define a **read-only automatic property** by omitting the set scope which can be set **only** in the **constructor**.
- However, it is **not possible** to define a **write only** auto property. Why?

Cont.

- By default, the hidden backing field is set to zero for numerical types, false for Boolean, and null for reference types.
- When using **automatic properties**, you **can initialize** the underlying backing field generated by the compiler like other fields **but** you **cannot** use this feature in **normal properties**.

Examples

```
private string gender;

public string Gender
{//normal property,
 //can have additional logic if necessary
    get { return gender; }
    set { gender = value; }
}
```

```
//auto-property, can't have any logic
public string Gender { get; set; }
```

```
//assigning value to the backing field,
//only allowed for auto-properties
public int MyProperty1 { get; set; } = 10;

//read-only auto-property,
//can only be assigned in a ctor
public int MyProperty2 { get;  }

//error,writeonly auto-prop. not allowed
//public int MyProperty3 { set; }

public Person()=> MyProperty2 = 1;
```

Object Initialization

- It is possible to create a new object and assign a lot of properties and fields in a few lines using **object initializer** syntax.
- Each member in the initialization list maps to the name of a public field or public property of the object being initialized.
- It's important to remember that this syntax is **using** the property **setter implicitly**(**must** be **public** to use this syntax).
- The **default constructor** will be called if there is **no parenthesis**.
- You can also call a **custom constructor** with this syntax but **make sure** you are **not** writing **duplicate** code.
- **init-Only** Setters (C# 9.0): Enables a property to have its value set **only** during **initialization**, **then** the **property** becomes **read-only**.
- These types of properties are called **immutable**.

Example

```
        //inside Main()
        Student s1 = new Student { Name = "Abebe", Age=26, Id=1170};
        Student s2 = new Student("Kebede", 25) { Id=1152 };
        //s1.Id = 9;//Error, becomes readonly after obj. creation
    }
}
class Student
{
    private string name;
    public string Name { get=>name; set { name = value; WriteLine("3"); } }
    public int Age { get; set; }
    public int Id { get; init; }
    public Student()
    {
        WriteLine("1");
    }
    public Student(string n,int a)
    {
        Name = n;
        Age = a;
        WriteLine("2");
    }
}
```

partial keyword

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- It is possible to **split** the definition of a **class**, a **struct**, an **interface,** or a **method** over two or more source files using the **partial** keyword **before** the keywords **class**, **struct**, **interface,** or the **return type**.
- Each section must be marked with the partial keyword and must be in the same namespace, the file name is irrelevant.

Example

```
public partial class Car
{
    public string carName="Audi";
    public partial string Print() //method implementation
    {
        int prodYear = 2023;
        WriteLine(prodYear);
        return engineType;
    }
}
public partial class Car
{
    private string engineType = "V8";
    public partial string Print();//method signature

}
```

Thank you