

# **.NET/C# Object-Oriented Programming (OOP).**

**Modern .NET/C# Course**

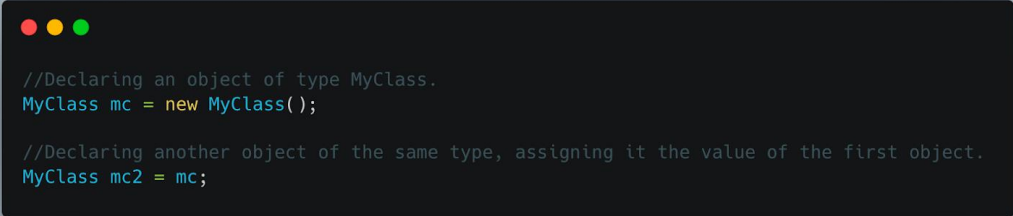
**.NET Academy**

# Content

- Class.
  - Properties (get, set, init).
  - Read-only properties.
  - Required properties.

# Classes.

A type that is defined as a class is a reference type. At run time, when you declare a variable of a reference type, the variable contains the value null until you explicitly create an instance of the class by using the new operator, or assign it an object of a compatible type that may have been created elsewhere, as shown in the following example:

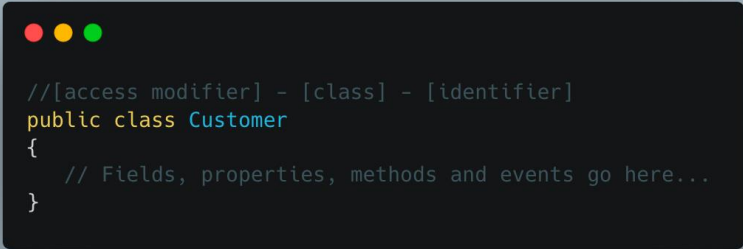


```
//Declaring an object of type MyClass.  
MyClass mc = new MyClass();  
  
//Declaring another object of the same type, assigning it the value of the first object.  
MyClass mc2 = mc;
```

# Classes.

When the object is created, enough memory is allocated on the managed heap for that specific object, and the variable holds only a reference to the location of said object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as garbage collection. However, garbage collection is also highly optimized and in most scenarios, it does not create a performance issue.

Classes are declared by using the class keyword followed by a unique identifier, as shown in the following example:



```
//[access modifier] - [class] - [identifier]
public class Customer
{
    // Fields, properties, methods and events go here...
}
```

# Classes.

The class keyword is preceded by the access level. Because public is used in this case, anyone can create instances of this class. The name of the class follows the class keyword. The name of the class must be a valid C# identifier name. The remainder of the definition is the class body, where the behavior and data are defined. Fields, properties, methods, and events on a class are collectively referred to as class members.

Although they are sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it is not an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

Objects can be created by using the new keyword followed by the name of the class that the object will be based on.


When an instance of a class is created, a reference to the object is passed back to the programmer.

In the previous example, object1 is a reference to an object that is based on Customer.

This reference refers to the new object but does not contain the object data itself. In fact, you can create an object reference without creating an object at all.

# Classes.

We don't recommend creating object references such as the preceding one that don't refer to an object because trying to access an object through such a reference will fail at run time. However, such a reference can be made to refer to an object, either by creating a new object.



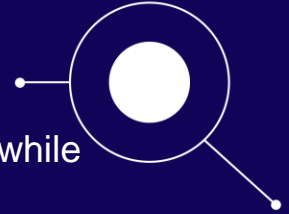
```
Customer object3 = new Customer();  
Customer object4 = object3;
```

This code creates two object references that both refer to the same object. Therefore, any changes to the object made through object3 are reflected in subsequent uses of object4. Because objects that are based on classes are referred to by reference, classes are known as reference types.

# Properties (get, set, init).

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they're public data members, but they're special methods called accessors. This feature enables data to be accessed easily and still helps promote the safety and flexibility of methods.

# Properties (get, set, init).



- Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
- A get property accessor is used to return the property value, and a set property accessor is used to assign a new value. In C# 9 and later, an init property accessor is used to assign a new value only during object construction. These accessors can have different access levels. For more information, see [Restricting Accessor Accessibility](#).
- The value keyword is used to define the value being assigned by the set or init accessor.
- Properties can be read-write (they have both a get and a set accessor), read-only (they have a get accessor but no set accessor), or write-only (they have a set accessor, but no get accessor). Write-only properties are rare and are most commonly used to restrict access to sensitive data.
- Simple properties that require no custom accessor code can be implemented either as expression body definitions or as auto-implemented properties.



# Properties (get, set, init).

One basic pattern for implementing a property involves using a private backing field for setting and retrieving the property value. The get accessor returns the value of the private field, and the set accessor may perform some data validation before assigning a value to the private field. Both accessors may also perform some conversion or computation on the data before it's stored or returned.

The following example illustrates this pattern. In this example, the `TimePeriod` class represents an interval of time. Internally, the class stores the time interval in seconds in a private field named `_seconds`. A read-write property named `Hours` allows the customer to specify the time interval in hours. Both the get and the set accessors perform the necessary conversion between hours and seconds. In addition, the set accessor validates the data and throws an `ArgumentOutOfRangeException` if the number of hours is invalid.

# Properties (get, set, init).

```
public class TimePeriod
{
    private double _seconds;

    public double Hours
    {
        get { return _seconds / 3600; }
        set
        {
            if (value < 0 || value > 24)
                throw new
ArgumentOutOfRangeException("value of {value}, between 0 and 24.");

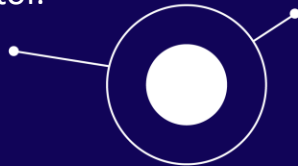
            _seconds = value * 3600;
        }
    }
}
```

# Properties (get, set, init).

In C# 9 and later, the `init` keyword defines an accessor method in a property or indexer. An `init`-only setter assigns a value to the property or the indexer element only during object construction. This enforces immutability, so that once the object is initialized, it can't be changed again.

The **`readonly`** keyword is a modifier that can be used in four contexts:

- In a field declaration, `readonly` indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class. A `readonly` field can be assigned and reassigned multiple times within the field declaration and constructor.



# Properties (get, set, init).

A readonly field can't be assigned after the constructor exits. This rule has different implications for value types and reference types:

- Because value types directly contain their data, a field that is a readonly value type is immutable.
- Because reference types contain a reference to their data, a field that is a readonly reference type must always refer to the same object. That object isn't immutable. The readonly modifier prevents the field from being replaced by a different instance of the reference type. However, the modifier doesn't prevent the instance data of the field from being modified through the read-only field.

**Warning** An externally visible type that contains an externally visible read-only field that is a mutable reference type may be a security vulnerability and may trigger warning CA2104 : "Do not declare read only mutable reference types."

- In a readonly struct type definition, readonly indicates that the structure type is immutable. For more information, see the readonly struct section of the Structure types article.
- In an instance member declaration within a structure type, readonly indicates that an instance member doesn't modify the state of the structure. For more information, see the readonly instance members section of the Structure types article.
- In a ref readonly method return, the readonly modifier indicates that method returns a reference and writes aren't allowed to that reference.

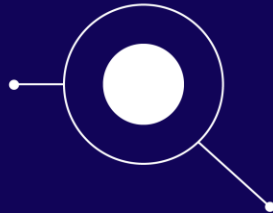
# Properties (get, set, init).

```
class Age
{
    private readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        // _year = 1967; // Compile error if
        // uncommented.
    }
}
```

# Required Modifier.

The required modifier indicates that the *field* or *property* it's applied to must be initialized by an object initializer. Any expression that initializes a new instance of the type must initialize all *required members*. The required modifier is available beginning with C# 11. The required modifier enables developers to create types where properties or fields must be properly initialized, yet still allow initialization using object initializers.

# Required Modifier.



Several rules ensure this behavior:

- The required modifier can be applied to *fields* and *properties* declared in struct, and class types, including record and record struct types. The required modifier can't be applied to members of an interface.
- Explicit interface implementations can't be marked as required. They can't be set in object initializers.
- Required members must be initialized, but they may be initialized to null. If the type is a non-nullable reference type, the compiler issues a warning if you initialize the member to null. The compiler issues an error if the member isn't initialized at all.
- Required members must be at least as visible as their containing type. For example, a public class can't contain a required field that's protected. Furthermore, required properties must have setters (set or init accessors) that are at least as visible as their containing types. Members that aren't accessible can't be set by code that creates an instance.

# Required Modifier.

- Derived classes can't hide a required member declared in the base class. Hiding a required member prevents callers from using object initializers for it. Furthermore, derived types that override a required property must include the required modifier. The derived type can't remove the required state. Derived types can add the required modifier when overriding a property.
- A type with any required members may not be used as a type argument when the type parameter includes the `new()` constraint. The compiler can't enforce that all required members are initialized in the generic code.
- The required modifier isn't allowed on the declaration for positional parameters on a record. You can add an explicit declaration for a positional property that does include the required modifier.



# Required Modifier.

Some types, such as positional records, use a primary constructor to initialize positional properties. If any of those properties include the required modifier, the primary constructor adds the `SetsRequiredMembers` attribute. This indicates that the primary constructor initializes all required members. You can write your own constructor with the `System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute` attribute. However, the compiler doesn't verify that these constructors do initialize all required members. Rather, the attribute asserts to the compiler that the constructor does initialize all required members. The `SetsRequiredMembers` attribute adds these rules to constructors:

- A constructor that chains to another constructor annotated with the `SetsRequiredMembers` attribute, either `this()`, or `base()`, must also include the `SetsRequiredMembers` attribute. That ensures that callers can correctly use all appropriate constructors.
- Copy constructors generated for record types have the `SetsRequiredMembers` attribute applied if any of the members are required.

# Required Modifier.

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName, string lastName) =>
        (FirstName, LastName) = (firstName, lastName);

    public required string FirstName { get; init; }
    public required string LastName { get; init; }

    public int? Age { get; set; }
}

public class Student : Person
{
    public Student() : base()
    {
    }

    [SetsRequiredMembers]
    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
    }

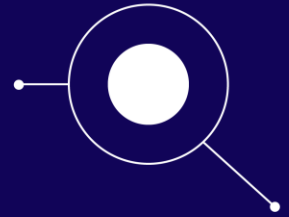
    public double GPA { get; set; }
}
```

# Questions for the study of the topic.



1. What is the fundamental difference between a class and an object in object-oriented programming?
2. When an instance of a class is created, what is stored in the variable, and how does memory allocation and reference assignment work in this context?
3. What is the purpose of properties in C#? How do they provide a flexible mechanism for working with data?
4. Distinguish between "get", "set" and "init" property accessors. How do these accessors affect the accessibility and modifiability of data in a class?
5. What is a "required modifier" introduced in C# 11 and where it can be applied in classes and structures?
6. Discuss the role of the "SetsRequiredMembers" attribute and its implications for types with mandatory members, especially in constructors.

# Homework



1. Explain the fundamental difference between a class and an object in object-oriented programming. Give examples illustrating this difference.
2. When an instance of a class is created, what is stored in the variable? Describe the process of allocating memory and assigning references.
3. Describe the concept of properties in C# and how they provide a flexible mechanism for working with data. Give examples of scenarios where properties can be useful.
4. Distinguish between "get", "set", and "init" property accessors. How do they affect the availability and modifiability of data?
5. What is the purpose of the "required modifier" in C# 11 and where can it be applied? Explain the rules and restrictions associated with the use of the required modifier.
6. Discuss the role of the "SetsRequiredMembers" attribute and its implications for types with required members. How does it affect the behaviour of the constructor?

**Thank you for your attention.**

**.NET Academy**