**C# Objects and Classes**

**Introduction to Object-Oriented Programming in C#**

C# is an object-oriented programming (OOP) language, which means that it is designed around the concept of objects and classes. This paradigm allows for the modeling of real-world entities and relationships, making it easier to manage and organize code.

**C# Object**

**Definition**

In C#, an **Object** is a real-world entity that has a distinct identity, state, and behavior. Examples of objects include:

Chair, Car, Pen, Mobile phone,Laptop

**Characteristics**

**State**: Represents the data or attributes of the object (e.g., color, size).

**Behavior**: Represents the functionality or methods of the object (e.g., drive, write).

**Runtime Entity**: Objects are created during the execution of a program (runtime).

**Instance of a Class**: An object is an instance of a class, meaning it can access all the members (fields, methods) defined in that class.

**Example of Creating an Object**

To create an object in C#, you typically use the new keyword followed by the class name. Here’s an **example:**

**Student s1 = new Student(); *// Creating an object of Student***

**In this example:**

**Student is the class type.**

s1 is the reference variable that holds the reference to the instance of the Student class.

The new keyword allocates memory for the object at runtime.

**C# Class**

**Definition**

In C#, a **Class** is a blueprint or template for creating objects. It defines the properties (fields) and behaviors (methods) that the objects created from the class will have.

**Example of a C# Class**

Here’s a simple example of a C# class that defines a Student with two fields:

**public class Student**

**{**

**int id; *// Field or data member***

**string name; *// Field or data member***

***// Constructor to initialize the fields***

**public Student(int studentId, string studentName)**

**{**

**id = studentId;**

**name = studentName;**

**}**

***// Method to display student details***

**public void DisplayInfo()**

**{**

**Console.WriteLine($"ID: {id}, Name: {name}");**

**}**

**}**

Key Components of the Class Example

**Fields**:

int id: Represents the student's ID.

string name: Represents the student's name.

**Constructor**:

public Student(int studentId, string studentName): Initializes the id and name fields when a new Student object is created.

**Method**:

public void DisplayInfo(): A method that prints the student's details to the console.

**C# Object and Class Example**

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

**using** System;

**public** **class** Student

    {

**int** id;//data member (also instance variable)

        String name;//data member(also instance variable)

**public** **static** **void** Main(**string**[] args)

        {

            Student s1 = **new** Student();//creating an object of Student

            s1.id = 101;

            s1.name = "Sonoo Jaiswal";

            Console.WriteLine(s1.id);

            Console.WriteLine(s1.name);

        }

    }

### C# Class Example 2: Having Main() in another class

Let's see another example of class where we are having Main() method in another class. In such case, class must be public.

**using** System;

**public** **class** Student

{

**public** **int** id;

**public** String name;

}

**class** TestStudent{

**public** **static** **void** Main(**string**[] args)

{

Student s1 = **new** Student();

s1.id = 101;

s1.name = "Sonoo Jaiswal";

Console.WriteLine(s1.id);

Console.WriteLine(s1.name);   }}}

# C# Constructor

In C#, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C# has the same name as class or struct.

There can be two types of constructors in C#.

* Default constructor
* Parameterized constructor

## C# Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

### C# Default Constructor Example: Having Main() within class

**using** System;

**public** **class** Employee

{

**public** Employee()

{

Console.WriteLine("Default Constructor Invoked");

}

**public** **static** **void** Main(**string**[] args)

{

Employee e1 = **new** Employee();

Employee e2 = **new** Employee();

}

}

### C# Default Constructor Example: Having Main() in another class

Let's see another example of default constructor where we are having Main() method in another class.

**using** System;

**public** **class** Employee

{

**public** Employee()

{

Console.WriteLine("Default Constructor Invoked");

}

}

**class** TestEmployee{

**public** **static** **void** Main(**string**[] args)

{

Employee e1 = **new** Employee();   //Default Constructor Invoked

Employee e2 = **new** Employee();  //Default Constructor Invoked

}

}

## C# Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

1. **using** System;
2. **public** **class** Employee
3. {
4. **public** **int** id;
5. **public** String name;
6. **public** **float** salary;
7. **public** Employee(**int** i, String n,**float** s)
8. {
9. id = i;
10. name = n;
11. salary = s;
12. }
13. **public** **void** display()
14. {
15. Console.WriteLine(id + " " + name+" "+salary);
16. }
17. }
18. **class** TestEmployee{
19. **public** **static** **void** Main(**string**[] args)
20. {
21. Employee e1 = **new** Employee(101, "Sonoo", 890000f);
22. Employee e2 = **new** Employee(102, "Mahesh", 490000f);
23. e1.display();   // 101 Sonoo 890000
24. e2.display();   // 102 Mahesh 490000
25. }
26. }

# Destructors in C#

## Introduction

In C#, a **Destructor** is a special method of a class used to destroy objects or instances of classes. Destructors are invoked automatically by the **Garbage Collector** when an instance of a class becomes unreachable, ensuring that resources are released properly.

### Key Properties of Destructors:

* **Class-Only Usage:** Destructors can only be defined in classes, not in structs.
* **Single Destructor per Class:** A class can contain only one destructor.
* **Tilde (~) Operator:** Destructors are represented using the tilde (~) operator followed by the class name.
* **No Parameters or Access Modifiers:** Destructors do not accept parameters and cannot have access modifiers.
* **Automatic Invocation:** The Garbage Collector automatically calls the destructor when the class instance is no longer needed.

## Syntax

The syntax for defining a destructor in a class is as follows:

***class User***

***{***

***// Destructor***

***~User()***

***{***

***// Your cleanup code here***

***}***

***}***

## Example

Below is an example demonstrating the use of a destructor alongside a constructor:

***using System;***

***class User***

***{***

***public User()***

***{***

***Console.WriteLine("An instance of the class created");***

***}***

***// Destructor***

***~User()***

***{***

***Console.WriteLine("An instance of the class destroyed");***

***}***

***}***

***class Program***

***{***

***static void Main(string[] args)***

***{***

***Details();***

***// Force garbage collection to demonstrate destructor invocation***

***GC.Collect();***

***GC.WaitForPendingFinalizers();***

***Console.ReadLine();***

***}***

***public static void Details()***

***{***

***// Create an instance of the class***

***User user = new User();***

***}***

***}***

### Output:

An instance of the class created

An instance of the class destroyed

### Explanation:

1. **Instance Creation:**
   * When the Details method is called, a new instance of the User class is created.
   * The **constructor** (public User()) is invoked, printing:  
     An instance of the class created
2. **Destructor Invocation:**
   * After the Details method completes, the user object goes out of scope and becomes eligible for garbage collection.
   * The GC.Collect() method is called to force garbage collection, ensuring that the destructor is invoked promptly.
   * The **destructor** (~User()) is then called by the Garbage Collector, printing:  
     An instance of the class destroyed
3. **Program Termination:**
   * The Console.ReadLine() statement keeps the console window open, allowing you to see the output.

--------------------------------------------------------------------------------------------------------------------------

# The this Keyword in C#

## Introduction

In C#, the this keyword is a special reference that points to the current instance of the class. It is commonly used to resolve naming conflicts, pass the current instance to other methods, and declare indexers. This document explores the three primary uses of the this keyword in C# with examples.

### Key Uses of the this Keyword:

1. **Referring to Current Class Instance Variables:**  
   The this keyword is often used to refer to the current class instance's fields or methods, especially when there is a naming conflict between instance variables and method parameters.
2. **Passing the Current Object as a Parameter:**  
   this can be used to pass the current object to another method or constructor, allowing for object chaining or further operations on the same instance.
3. **Declaring Indexers:**  
   The this keyword is used to define indexers in classes or structs, allowing objects to be indexed in a similar way to arrays.

## Example: Referring to Current Class Instance Variables

Below is an example that demonstrates how the this keyword is used to distinguish between instance variables and method parameters when they have the same name.

### Code Example

***using System;***

***public class Employee***

***{***

***public int id;***

***public string name;***

***public float salary;***

***// Constructor using 'this' to refer to instance variables***

***public Employee(int id, string name, float salary)***

***{***

***this.id = id; // 'this.id' refers to the instance variable, while 'id' refers to the constructor parameter***

***this.name = name;***

***this.salary = salary;***

***}***

***// Method to display employee details***

***public void display()***

***{***

***Console.WriteLine(id + " " + name + " " + salary);***

***}***

***}***

***class TestEmployee***

***{***

***public static void Main(string[] args)***

***{***

***// Creating instances of Employee***

***Employee e1 = new Employee(101, "Sonoo", 890000f);***

***Employee e2 = new Employee(102, "Mahesh", 490000f);***

***// Displaying the details of each employee***

***e1.display();***

***e2.display();***

***}***

***}***

### Output:

101 Sonoo 890000

102 Mahesh 490000

### Explanation:

* **Constructor Usage:**  
  The Employee class constructor accepts parameters id, name, and salary. The this keyword is used to refer to the instance variables (this.id, this.name, and this.salary) to differentiate them from the parameters with the same names.
* **Object Creation:**  
  Two instances of the Employee class are created with different values, and the display() method is called to print the details of each employee.
* **Display Method:**  
  The display() method outputs the id, name, and salary for each Employee instance, showing how the this keyword correctly assigns values to the instance variables.

# The static Keyword in C#

## Introduction

In C#, the static keyword is used to indicate that a member belongs to the type itself, rather than to any instance of the type. This means that static members are shared across all instances of a class and can be accessed without creating an object of the class. The static keyword can be applied to fields, methods, constructors, classes, properties, operators, and events.

### Key Characteristics of static Members:

* **Type-Level Association:** Static members are associated with the type itself, not with any specific object instance.
* **No Instance Required:** Static members can be accessed without creating an instance of the class.
* **Memory Efficiency:** Since static members are shared among all instances, they help save memory by avoiding duplication of the same data across multiple instances.

### Important Notes:

* **Non-Static Members:** Indexers and destructors cannot be declared as static.
* **Singleton Behavior:** Static fields and methods can be used to implement the Singleton design pattern, ensuring that only one instance of a class is created.

## Advantages of the static Keyword

### Memory Efficiency

Static members do not require an instance of the class to be accessed. This characteristic saves memory since the static members are shared across all instances of the class. As a result, the memory footprint of static members is significantly smaller compared to instance members, especially in scenarios with many instances.

## Static Field

A **static field** is a variable that is shared across all instances of a class. Unlike instance fields, which get memory allocated every time an object is created, a static field is allocated memory only once and is shared among all objects of the class.

### Example: Static Field in C#

*using System;*

*class User*

*{*

*// Static Variables*

*public static string name, location;*

*// Non-Static Variable*

*public int age;*

*// Non-Static Method*

*public void Details()*

*{*

*Console.WriteLine("Non-Static Method");*

*}*

*// Static Method*

*public static void Details1()*

*{*

*Console.WriteLine("Static Method");*

*}*

*}*

*class Program*

*{*

*static void Main(string[] args)*

*{*

*// Creating an instance of the User class*

*User u = new User();*

*// Assigning value to non-static variable*

*u.age = 32;*

*// Calling non-static method*

*u.Details();*

*// Assigning values to static variables*

*User.name = "Suresh Dasari";*

*User.location = "Hyderabad";*

*// Displaying the values of static and non-static variables*

*Console.WriteLine("Name: {0}, Location: {1}, Age: {2}", User.name, User.location, u.age);*

*// Calling static method*

*User.Details1();*

*Console.WriteLine("\nPress Enter Key to Exit..");*

*Console.ReadLine();*

*}*

*}*

### Output:

Non-Static Method

Name: Suresh Dasari, Location: Hyderabad, Age: 32

Static Method

## Explanation

### Static Members:

* **Static Variables (name, location)**:  
  These variables are shared among all instances of the User class. They are assigned and accessed using the class name (User.name, User.location) without the need for an object instance.
* **Static Method (Details1())**:  
  This method can be called directly using the class name (User.Details1()), without requiring an instance of the User class.

### Non-Static Members:

* **Non-Static Variable (age)**:  
  This variable is specific to each instance of the User class. In this example, it is assigned and accessed using the object instance u.
* **Non-Static Method (Details())**:  
  This method operates on the instance-specific data and must be called using an object instance (u.Details()).

### Summary of Execution:

1. A User object u is created, and the non-static variable age is set to 32.
2. The non-static method Details() is called, which prints "Non-Static Method".
3. Static variables name and location are assigned values and accessed directly via the User class.
4. The values of name, location, and age are printed to the console.
5. The static method Details1() is called, which prints "Static Method".

----------------------------------------------------------------------------------------------------------------

# C# Static Class

## Introduction

A static class in C# is similar to a regular class, but it cannot be instantiated. This means that you cannot create an object of a static class. The primary purpose of a static class is to hold static members that are globally accessible throughout the application.

## Key Points

* **Only Static Members:** A static class can only contain static members (fields, methods, properties, etc.).
* **Cannot Be Instantiated:** You cannot create an instance of a static class.
* **Sealed Class:** A static class is implicitly sealed, meaning it cannot be inherited.
* **No Instance Constructors:** A static class cannot have instance constructors; it only has a static constructor (if needed).

## Example

Below is an example of a static class that contains a static field and a static method.

***using System;***

***public static class MyMath***

***{***

***// Static Field***

***public static float PI = 3.14f;***

***// Static Method***

***public static int cube(int n)***

***{***

***return n \* n \* n;***

***}***

***}***

***class TestMyMath***

***{***

***public static void Main(string[] args)***

***{***

***// Accessing static field***

***Console.WriteLine("Value of PI is: " + MyMath.PI);***

***// Accessing static method***

***Console.WriteLine("Cube of 3 is: " + MyMath.cube(3));***

***}***

*}*

### Output

Value of PI is: 3.14

Cube of 3 is: 27

## Explanation

### Static Field (PI)

* The field PI is a static field within the MyMath class. Since it is static, it belongs to the class itself rather than any instance of the class. It is accessible directly using the class name (MyMath.PI).

### Static Method (cube)

* The method cube(int n) is a static method that calculates the cube of a given integer. Like the static field, it is accessed using the class name (MyMath.cube(3)).

### Accessing Static Members

* Since MyMath is a static class, we directly use the class name to access its members. There’s no need to create an instance of the class to use PI or cube.

## Advantages of Using Static Classes

* **Memory Efficiency:** Since static classes cannot be instantiated, they avoid unnecessary memory usage associated with object creation.
* **Global Accessibility:** Static members are globally accessible within the application, making them convenient for utility functions and constants.

------------------------------------------------------------------------------------------------------------------------

### C# Static Constructor Overview

A **static constructor** in C# is designed to perform actions that need to occur only once during the application's lifecycle. It is called automatically before the first instance of a class is created or any static members are accessed. This ensures that the necessary setup is done before the class is used.

Here are the key characteristics of a static constructor in C#:

1. **No Parameters and Access Modifiers**:
   * Static constructors do not accept parameters.
   * They cannot have access modifiers like public, private, etc. They are implicitly private.
2. **Automatic Invocation**:
   * The static constructor is invoked automatically by the CLR (Common Language Runtime) before any static members or the first instance of the class are accessed.
   * You cannot manually call a static constructor.
3. **Execution Only Once**:
   * A static constructor is called only once during the entire application lifetime, ensuring that the initialization is performed only once.
4. **No Control Over Execution Order**:
   * The execution order of static constructors in different classes is determined by the CLR. Developers do not have control over when a static constructor is executed relative to other static constructors.
5. **Single Static Constructor Per Class**:
   * A class can have only one static constructor. Having multiple static constructors is not allowed.

***using System;***

***namespace Tutlane***

***{***

***class User***

***{***

***// Static Constructor***

***static User()***

***{***

***Console.WriteLine("I am Static Constructor");***

***}***

***// Default Constructor***

***public User()***

***{***

***Console.WriteLine("I am Default Constructor");***

***}***

***}***

***class Program***

***{***

***static void Main(string[] args)***

***{***

***// Both Static and Default constructors will invoke for the first instance***

***User user = new User();***

***// Only the Default constructor will invoke***

***User user1 = new User();***

***Console.WriteLine("\nPress Enter Key to Exit..");***

***Console.ReadLine();***

***}***

***}***

***}***

### ---------------------------------------------------------------------------------------------------------------------------------------C# Structures (Structs)

In C#, structures (structs) are similar to classes but with a key difference: classes are reference types, while structures are value types. As value types, structures directly store their values, meaning their instances are stored on the stack, making them generally faster than classes.

Structures in C# can contain various members, including fields, properties, methods, operators, constructors, events, indexers, constants, and even other structures.

### Example of a C# Structure

Below is an example demonstrating how to create a structure in C# with various fields and a parameterized constructor.

***using System;***

***namespace Tutlane***

***{***

***struct User***

***{***

***public const string name = "Suresh Dasari";***

***public string location;***

***public int age;***

***// Parameterized Constructor***

***public User(string location, int age)***

***{***

***this.location = location;***

***this.age = age;***

***}***

***}***

***class Program***

***{***

***static void Main(string[] args)***

***{***

***// Creating an instance using the new keyword***

***User u = new User("Hyderabad", 31);***

***// Creating an instance without using the new keyword***

***User u1;***

***// Displaying the values of the first user***

***Console.WriteLine("Name: {0}, Location: {1}, Age: {2}", User.name, u.location, u.age);***

***// Initializing fields for the second user***

***u1.location = "Guntur";***

***u1.age = 32;***

***Console.WriteLine("Name: {0}, Location: {1}, Age: {2}", User.name, u1.location, u1.age);***

***Console.WriteLine("\nPress Enter to Exit..");***

***Console.ReadLine();***

***}***

***}***

***}***

### Characteristics of C# Structures

Here are some important characteristics of structures in C#:

* **Value Type:** Structures are value types and are defined using the struct keyword.
* **Field Initialization:** Fields cannot be initialized within a structure unless they are declared as const or static.
* **Members:** Structures can include fields, properties, methods, operators, constructors, events, indexers, constants, and other structures.
* **Constructors:** Structures cannot have a default (parameterless) constructor or a destructor, but they can have parameterized constructors.
* **Inheritance:** Structures cannot inherit from another structure or class.
* **Interfaces:** Structures can implement interfaces.
* **Instantiation:** Structures can be instantiated with or without the new keyword.

### Differences Between Structures and Classes in C#

Below are the key differences between structures and classes:

* **Type:** Classes are reference types, whereas structures are value types.
* **Constructors:** Classes can have both default constructors and destructors, while structures can only have constructors with parameters.
* **Inheritance:** Classes support inheritance, but structures do not.
* **Instantiation:** Unlike classes, structures can be instantiated either with or without the new operator.

------------------------------------------------------------------------------------------------------------------------------------

### C# Enum

In C#, an enum (short for enumeration) is a special data type used to define a set of named constants. Enums are particularly useful when you need to represent a collection of related values, such as seasons, days of the week, months, or sizes. The constants within an enum are referred to as enumerators.

Enums can be declared either within or outside classes and structs, providing flexibility in how they are utilized.

By default, the values of enum constants start at 0 and increment by 1 for each subsequent member. However, you can explicitly set the values if needed.

### Key Points About Enums

* **Fixed Set of Constants:** Enums have a fixed set of named constants.
* **Type Safety:** Enums enhance type safety by limiting the set of possible values for a variable.
* **Traversable:** Enums can be easily traversed, making them useful for iteration.

### Simple C# Enum Example

Here is a simple example demonstrating the use of enums in C#:

***using System;***

***public class EnumExample***

***{***

***// Declaring an enum for Seasons***

***public enum Season { WINTER, SPRING, SUMMER, FALL }***

***public static void Main()***

***{***

***// Casting enum values to their underlying integer types***

***int x = (int)Season.WINTER;***

***int y = (int)Season.SUMMER;***

***// Displaying the integer values of the enum members***

***Console.WriteLine("WINTER = {0}", x);***

***Console.WriteLine("SUMMER = {0}", y);***

***}***

***}***

**Output:**

WINTER = 0

SUMMER = 2

### C# Enum Example with Custom Start Index

You can customize the starting value of enum members. In the following example, the Season enum is set to start with a custom value:

***using System;***

***public class EnumExample***

***{***

***// Declaring an enum with a custom starting value***

***public enum Season { WINTER = 10, SPRING, SUMMER, FALL }***

***public static void Main()***

***{***

***// Casting enum values to their underlying integer types***

***int x = (int)Season.WINTER;***

***int y = (int)Season.SUMMER;***

***// Displaying the integer values of the enum members***

***Console.WriteLine("WINTER = {0}", x);***

***Console.WriteLine("SUMMER = {0}", y);***

***}***

***}***

***using System;***

***public class EnumExample***

***{***

***// Declaring an enum for Days of the Week***

***public enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat }***

***public static void Main()***

***{***

***// Traversing the enum using GetNames() method***

***Console.WriteLine("Traversing using Enum.GetNames():");***

***foreach (string name in Enum.GetNames(typeof(Days)))***

***{***

***Console.WriteLine(name);***

***}***

***Console.WriteLine();***

***// Traversing the enum using GetValues() method***

***Console.WriteLine("Traversing using Enum.GetValues():");***

***foreach (Days day in Enum.GetValues(typeof(Days)))***

***{***

***Console.WriteLine(day);***

***}***

***}***

***}***

**Output:**

Traversing using Enum.GetNames():

Sun

Mon

Tue

Wed

Thu

Fri

Sat

Traversing using Enum.GetValues():

Sun

Mon

Tue

Wed

Thu

Fri

Sat

### Explanation:

* **Enum.GetNames(typeof(Days)):** This method retrieves the names of the enum members as strings, which are then printed one by one.
* **Enum.GetValues(typeof(Days)):** This method retrieves the enum members themselves, allowing you to iterate through each value and print it.

This combined example shows how to traverse an enum using both GetNames() and GetValues() methods in C#.

---------------------------------------------------------------------------------------------------------------------------------------

### Properties (GET, SET) in C#

In C#, a property is an extension of a class variable that provides a controlled mechanism for reading, writing, or changing the value of a class variable. Properties help to encapsulate the internal state of an object while exposing it in a safe and controlled manner to the outside world.

Properties typically include two code blocks known as accessors:

* **Get Accessor:** Used to return the value of the property.
* **Set Accessor:** Used to assign a new value to the property.

Using properties, you can control how a class's internal data is accessed and modified, allowing for validations and other logic to be implemented without affecting the external interface.

### Syntax for Defining a Property in C#

***<access\_modifier> <return\_type> <property\_name>***

***{***

***get***

***{***

***// Return the property value***

***}***

***set***

***{***

***// Set a new value***

***}***

***}***

### Example of Properties with GET and SET Accessors

Below is an example that demonstrates how to define properties in C# with both get and set accessors, including validations:

***using System;***

***namespace Tutlane***

***{***

***class User***

***{***

***// Private fields***

***private string location;***

***private string name = "Suresh Dasari";***

***// Public property for Location***

***public string Location***

***{***

***get { return location; }***

***set { location = value; }***

***}***

***// Public property for Name with custom logic***

***public string Name***

***{***

***get***

***{***

***// Return the name in uppercase***

***return name.ToUpper();***

***}***

***set***

***{***

***// Set the name only if the value is "Suresh"***

***if (value == "Suresh")***

***name = value;***

***}***

***}***

***}***

***class Program***

***{***

***static void Main(string[] args)***

***{***

***// Create an instance of the User class***

***User u = new User();***

***// Invoke the set accessor to change the name***

***u.Name = "Rohini";***

***// Invoke the set accessor to change the location***

***u.Location = "Hyderabad";***

***// Invoke the get accessor to retrieve and print the name***

***Console.WriteLine("Name: " + u.Name);***

***// Invoke the get accessor to retrieve and print the location***

***Console.WriteLine("Location: " + u.Location);***

***Console.WriteLine("\nPress Enter Key to Exit..");***

***Console.ReadLine();***

***}***

***}***

***}***

### Explanation:

* **Name Property:**
  + The get accessor returns the name in uppercase.
  + The set accessor only changes the name if the value is "Suresh". If you try to set the name to something else, it won’t change the original value.
* **Location Property:**
  + The get accessor simply returns the value of the location field.
  + The set accessor allows you to assign any value to the location field.

### Output:

When you run the above code, it will produce the following output:

Name: SURESH DASARI

Location: Hyderabad

In this example, although the set accessor for Name was called with "Rohini", the internal validation prevented the change, so the get accessor still returns "SURESH DASARI". The Location property, however, successfully updates and retrieves the value "Hyderabad

### C# Auto-Implemented Properties

In C#, a property is referred to as an **auto-implemented property** when it contains only the get and set accessors without any additional logic. Auto-implemented properties provide a simple and concise way to declare properties when no additional logic is required in the accessors.

Auto-implemented properties are particularly useful when you just need to store and retrieve values without any additional processing or validation.

### Example of Auto-Implemented Properties in C#

Here's an example demonstrating the use of auto-implemented properties in C#:

***using System;***

***namespace Tutlane***

***{***

***class User***

***{***

***// Auto-implemented properties***

***public string Name { get; set; }***

***public string Location { get; set; }***

***}***

***class Program***

***{***

***static void Main(string[] args)***

***{***

***// Create an instance of the User class***

***User u = new User();***

***// Assign values to properties***

***u.Name = "Suresh Dasari";***

***u.Location = "Hyderabad";***

***// Access and print the property values***

***Console.WriteLine("Name: " + u.Name);***

***Console.WriteLine("Location: " + u.Location);***

***Console.WriteLine("\nPress Enter Key to Exit..");***

***Console.ReadLine();***

***}***

***}***

***}***

### Explanation:

* **Auto-Implemented Properties:** The Name and Location properties are declared with only get and set accessors. No additional logic is provided, making them auto-implemented properties.
* **Usage:** The properties are directly accessed and modified using the dot notation (u.Name, u.Location), making the code clean and easy to read.

### C# Properties Overview

Here are some important points to remember about properties in C#:

* **Public Exposure:** Properties allow class variables to be exposed in a public manner using get and set accessors while hiding the implementation details.
* **Accessors:**
  + **Get Accessor:** Used to return the value of the property.
  + **Set Accessor:** Used to assign a new value to the property. The value keyword represents the value being assigned.
* **Property Categories:**
  + **Read-Write Property:** Contains both get and set accessors.
  + **Read-Only Property:** Contains only a get accessor, preventing modification.
  + **Write-Only Property:** Contains only a set accessor, preventing reading.

**-**-------------------------------------------------------------------------------------------------------------------------

### C# readonly vs const

In C#, both readonly and const are used to define fields whose values are not supposed to change after being initialized. However, they have different characteristics and use cases.

### 1. const in C#

* **Definition:** A const field is a constant and its value is set at compile-time. Once defined, its value cannot be changed.
* **Initialization:** Must be initialized at the time of declaration.
* **Scope:** const fields are implicitly static, meaning they belong to the type itself rather than to any instance of the type.
* **Value Assignment:** Can only be assigned a value that is known at compile-time (e.g., literals, other constants).

### Example of const:

***using System;***

***class Example***

***{***

***// Const field***

***public const double Pi = 3.14159;***

***static void Main(string[] args)***

***{***

***// Accessing const field***

***Console.WriteLine("Value of Pi: " + Pi);***

***// Pi = 3.14; // Error: Cannot assign to 'Pi' because it is a 'const'***

***Console.ReadLine();***

***}***

***}***

### 2. readonly in C#

* **Definition:** A readonly field can only be assigned once, either at the time of declaration or in the constructor of the class. Unlike const, it can hold values that are determined at runtime.
* **Initialization:** Can be initialized at the time of declaration or within the constructor.
* **Scope:** readonly fields can be instance-level or static.
* **Value Assignment:** Can be assigned values that are determined at runtime, which provides more flexibility than const.

### Example of readonly:

***using System;***

***class Example***

***{***

***// Readonly field***

***public readonly double Pi;***

***// Constructor***

***public Example(double piValue)***

***{***

***Pi = piValue;***

***}***

***static void Main(string[] args)***

***{***

***// Creating instances with different values of Pi***

***Example example1 = new Example(3.14159);***

***Example example2 = new Example(3.14);***

***// Accessing readonly fields***

***Console.WriteLine("Value of Pi in example1: " + example1.Pi);***

***Console.WriteLine("Value of Pi in example2: " + example2.Pi);***

***// example1.Pi = 3.15; // Error: Cannot assign to 'Pi' because it is a 'readonly' field***

***Console.ReadLine();***

***}***

***}***

### Key Differences Between readonly and const:

1. **Initialization:**
   * const: Must be initialized at the time of declaration.
   * readonly: Can be initialized at the time of declaration or within the constructor.
2. **Value Assignment:**
   * const: Can only hold compile-time constant values.
   * readonly: Can hold runtime-determined values.
3. **Scope:**
   * const: Implicitly static, meaning it belongs to the type itself.
   * readonly: Can be either static or instance-level.
4. **Usage Context:**
   * const: Use when the value is constant and unchanging for all instances.
   * readonly: Use when the value might be different for different instances and needs to be determined at runtime.

### Summary:

* Use const for values that are fixed and known at compile-time, such as mathematical constants or fixed strings.
* Use readonly for values that need to be assigned once but may differ for each instance of a class or can only be known at runtime.