Table of Contents

[C# Notes 2](#_Toc176980802)

[Escape Sequences in C# 2](#_Toc176980803)

[In C# types 2](#_Toc176980804)

[Value Types 2](#_Toc176980805)

[Reference Types 2](#_Toc176980806)

[Null Coalescing Operator (??): 3](#_Toc176980807)

[When to Use the Null Coalescing Operator 3](#_Toc176980808)

[Datatype conversions 5](#_Toc176980809)

[Method Parameters 7](#_Toc176980810)

[Parameter Reference 7](#_Toc176980811)

[Out Parameter: 8](#_Toc176980812)

[Parameter Array: 9](#_Toc176980813)

[Static Constructors: 10](#_Toc176980814)

[Calling Parametrized Constructor from Default 12](#_Toc176980815)

[Inheritance in C# 13](#_Toc176980816)

[Method Hiding 14](#_Toc176980817)

[Polymorphism 16](#_Toc176980818)

[Properties 18](#_Toc176980819)

[Class Vs Struct 20](#_Toc176980820)

[Interfaces 20](#_Toc176980821)

[Abstract Classes 21](#_Toc176980822)

[Delegates 23](#_Toc176980823)

[Exception Handling 28](#_Toc176980824)

[Inner Exceptions in C# 29](#_Toc176980825)

[Why Use Inner Exceptions? 29](#_Toc176980826)

[How to Use Inner Exceptions 29](#_Toc176980827)

[Enum 30](#_Toc176980828)

[Regions 32](#_Toc176980829)

[Access Modifiers 33](#_Toc176980830)

[Attributes 33](#_Toc176980831)

[Reflection 34](#_Toc176980832)

[Generics 39](#_Toc176980833)

# C# Notes

Escape Sequences in C#  
<http://msdn.microsoft.com/en-us/library/h21280bw.aspx>  
Verbatim Literal is a string with an @ symbol prefix, as in @“Hello".   
Verbatim literals make escape sequences translate as normal printable characters to enhance readability.   
Practical Example:  
Without Verbatim Literal : “C:\\Pragim\\DotNet\\Training\\Csharp” – Less Readable  
With Verbatim Literal : @“C:\Pragim\DotNet\Training\Csharp” – Better Readable

In C# types   
Value Types  - int, float, double, structs, enums etc  
Reference Types – Interface, Class, delegates, arrays etc

## Value Types

* **Independent Copies:** Each variable holds its own copy.
* **Examples:** int, float, bool, char, struct
* **Behavior:** Changing one variable does not affect the other.
* **Memory:** Stored in the stack.
* int a = 10;
* int b = a; // b gets a copy of a
* b = 20; // Changing b doesn't affect a
* Console.WriteLine(a); // Outputs 10

## Reference Types

* **Shared Access:** Variables point to the same object.
* **Examples:** class, array, string, object
* **Behavior:** Changing one variable affects the other.
* **Memory:** Stored in the heap, reference stored in the stack.
* class Person
* {
* public string Name;
* }
* Person p1 = new Person();
* p1.Name = "Alice";
* Person p2 = p1; // p2 references the same Person object
* p2.Name = "Bob"; // Changing p2 affects p1
* Console.WriteLine(p1.Name); // Outputs "Bob"

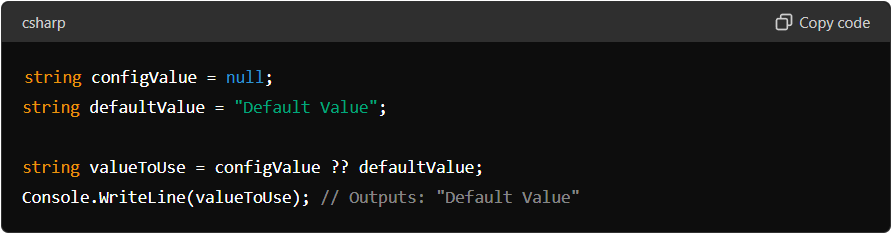
***By default value types are non nullable. To make them nullable use ?***  
int i = 0 (i is non nullable, so "i" cannot be set to null, i = null will generate compiler error)  
int? j = 0 (j is nullable int, so j=null is legal)

## Null Coalescing Operator (??):

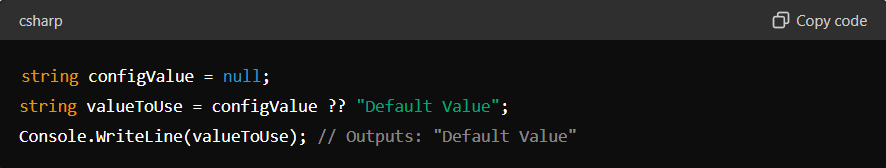
* Checks if the first value is null. If it is, it uses the second value.
* The null coalescing operator (??) in C# is a useful tool for handling situations where a variable might be null. It helps ensure that you have a fallback value when dealing with potential nulls, which can prevent errors and make your code more robust and readable.

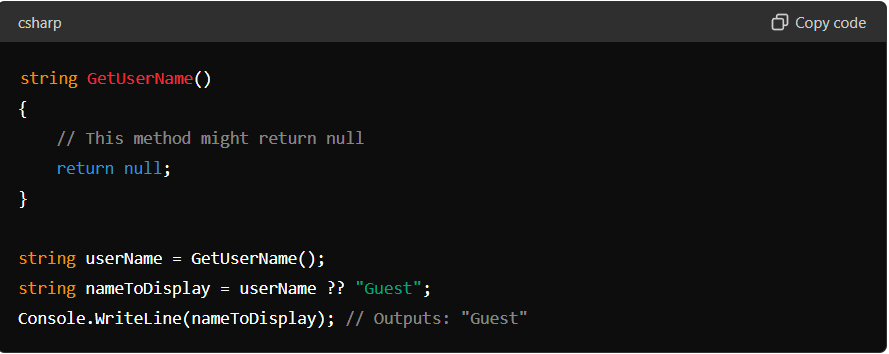
## When to Use the Null Coalescing Operator

* **Default Values:** When you want to provide a default value if a variable is null.



* **Simplifying Code:** It can make your code cleaner and shorter by reducing the need for explicit null checks.



* **Avoiding Null Reference Exceptions:** By using a fallback value, you can avoid errors that occur when you try to use a null object.  
  

**Practical Scenarios**

1. **User Inputs:** When you handle user input that might be empty or null, and you want to provide a default response.
2. **Configuration Settings:** When reading settings or configuration values that might be missing or null.
3. **Database Values:** When retrieving values from a database where some fields might be null.

***Nullable types bridge the differences between C# types and Database types***

**Program without using NULL coalescing operator**  
using System;  
class Program  
{  
    static void Main()  
    {  
        int AvailableTickets;  
        int? TicketsOnSale = null;  
  
        if (TicketsOnSale == null)  
        {  
            AvailableTickets = 0;  
        }  
        else  
        {  
            AvailableTickets = (int)TicketsOnSale;  
        }  
  
        Console.WriteLine("Available Tickets={0}", AvailableTickets);  
    }  
}  
  
**The above program is re-written using NULL coalescing operator**  
using System;  
class Program  
{  
    static void Main()  
    {  
        int AvailableTickets;  
        int? TicketsOnSale = null;  
  
        //Using null coalesce operator ??  
        AvailableTickets = TicketsOnSale ?? 0;  
  
        Console.WriteLine("Available Tickets={0}", AvailableTickets);  
    }  
}

## Datatype conversions

**Implicit conversion is done by the compiler:**  
**1.** When there is no loss of information if the conversion is done  
**2.** If there is no possibility of throwing exceptions during the conversion

**Example:**Converting an **int** to a **float** will not loose any data and no exception will be thrown, hence an implicit conversion can be done.   
Where as when converting a **float** to an **int**, we loose the fractional part and also a possibility of overflow exception. Hence, in this case an explicit conversion is required. For explicit conversion we can use cast operator or the convert class in c#.

**Explicit Conversion Example**  
using System;  
class Program  
{  
    public static void Main()  
    {  
        float f = 100.25F;  
  
        // Cannot implicitly convert float to int.  
        // Fractional part will be lost. Float is a  
        // bigger datatype than int, so there is  
        // also a possiblity of overflow exception  
        // int i = f;  
  
        // Use explicit conversion using cast () operator  
        int i = (int)f;  
  
        // OR use Convert class  
        // int i = Convert.ToInt32(f);  
  
        Console.WriteLine(i);  
    }  
}

**Difference between cast operator and Convert class:**

Whenever if there is error converting one data type to another, using simply cast operator it does not throws any exception but where as convert class throws an exception.

Example:

using System;

public class HelloWorld

{

public static void Main(string[] args)

{

float f = 1233546789654893.25f;

//Tryig to convert using cast doesn't throws exception

//It just displays lowest int value. i.e -2147483648

int i = (int)f;

//Using convert class gives out of flow exception

int j = Convert.ToInt32(f);

}

}

**Difference between Parse and TryParse**  
**1.** If the number is in a string format you have 2 options - Parse() and TryParse()   
**2.** Parse() method throws an exception if it cannot parse the value, whereas TryParse() returns a bool indicating whether it succeeded or failed.

using System;

public class HelloWorld

{

public static void Main(string[] args)

{

string demo = "100ff";

//Using parse methos it throws error if string to be converted in not a proper number

int num = int.Parse(demo);

//If we use Try Parse it returns true or false value

int result = 0;

int.TryParse(demo,out result);

}

}

## Method Parameters

## Parameter Reference

In C#, reference parameters allow a method to modify the variable passed to it. When you pass a parameter by reference, the method receives a reference to the original variable, not a copy of its value. This means that any changes made to the parameter inside the method will affect the original variable.

You use the ref keyword to pass parameters by reference. Here's an example:



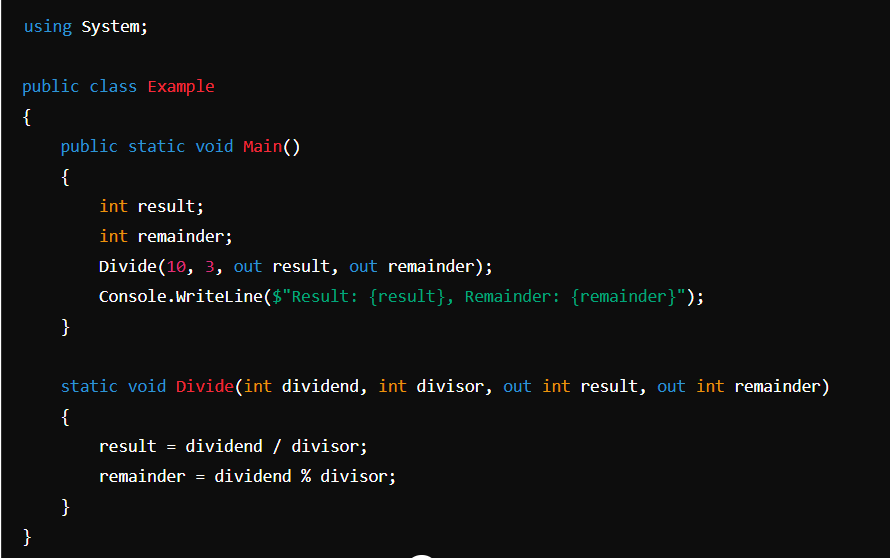
*Practical Scenarios:*

1. Managing State Between Methods: When dealing with session state, query strings, or form data, you might need to pass and update state across various methods. Using reference parameters ensures changes are consistently reflected.
2. Handling Large Data Models: When working with large data models, passing them by reference can improve performance by avoiding the overhead of copying data.
3. Database Operations: Reference parameters can be used in methods that perform complex database operations, such as inserting, updating, or deleting records, where you might need to return multiple updated values or statuses.
4. Caching and Optimization: When implementing caching mechanisms, reference parameters can help manage cached objects efficiently, ensuring that the same instance is used and updated across different parts of the application.
5. Service Layer Methods: In a service layer, methods often need to update multiple parameters or objects, making reference parameters useful for maintaining consistency and reducing code complexity.
6. Error Handling and Logging: Methods that handle errors or log activities might need to update the status or message objects passed to them, ensuring that the calling method gets the updated information.

## Out Parameter:

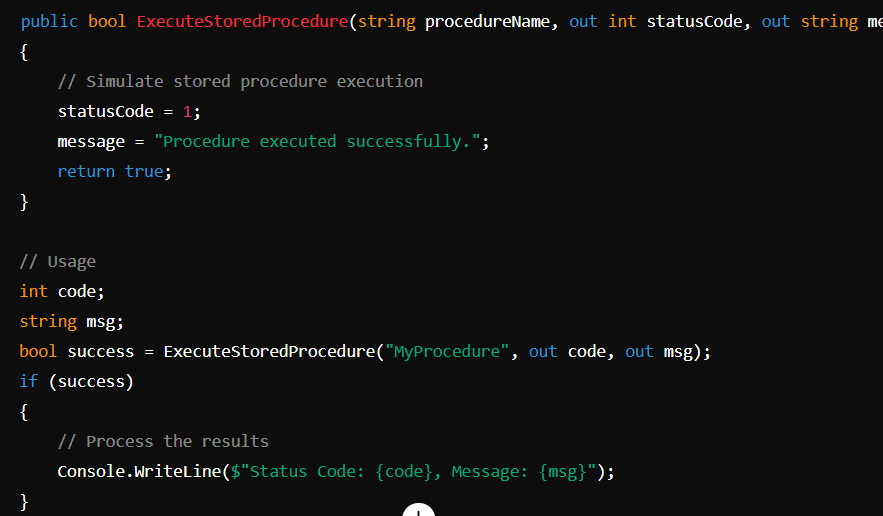
In C#, the out keyword is used to pass arguments to methods by reference, allowing the method to return multiple values. Unlike ref parameters, out parameters do not need to be initialized before they are passed to the method, but they must be assigned a value before the method returns.

Syntax and Example:



*Practical Scenarios:*

1. Returning Multiple Values from a Method
2. Database Operations: When working with databases, you might encounter situations where you need to return multiple values from a stored procedure or a query. For example, a method might execute a stored procedure that returns a status code and a message:



1. Parsing Methods: Parsing methods often use out parameters to return parsed values. For example, int.TryParse attempts to parse a string into an integer and returns a boolean indicating success or failure:

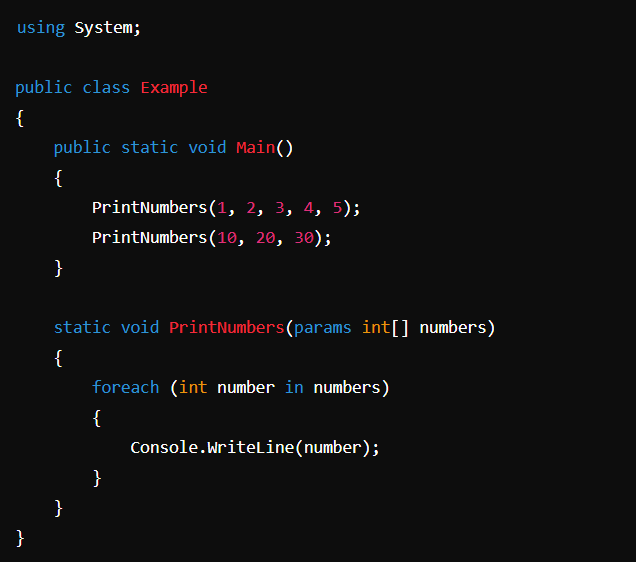
## Parameter Array:

In C#, parameter arrays, also known as params arrays, allow a method to accept a variable number of arguments. This is particularly useful when the exact number of arguments is unknown or when you want to provide a more flexible API.

**Syntax and Example**

The params keyword is used to specify a parameter array. Only one params keyword is allowed per method, and it must be the last parameter in the method signature.

Here’s a basic example demonstrating the use of params:



**Practical Scenarios in ASP.NET**

**1. Logging Utility:** A common use case for params is creating flexible logging methods that can accept any number of messages or parameters.

**2. Parameterized Queries**

In ASP.NET applications, you might use params to create methods that build SQL queries with a variable number of parameters.

## Static Constructors:

In C#, a static constructor is a special constructor that is used to initialize static members of a class. It is called automatically before any static members are accessed or any static methods are called. Here are some key points about static constructors:

1. **No Access Modifiers**: Static constructors cannot have access modifiers and cannot be called directly.
2. **Single Instance**: They are called only once for any given type, ensuring that static members are initialized only once.
3. **No Parameters**: Static constructors cannot take parameters.
4. **Automatic Invocation**: The runtime calls the static constructor automatically when the class is first accessed.

*Syntax:*

public class MyClass

{ static MyClass() {

// Initialization code here

} }

**Practical Scenarios in ASP.NET**

1. **Configuration Initialization**: Imagine you have some settings that you need to read from a configuration file only once, like a connection string to your database. You can use a static constructor to load this information when your application first starts.

public class ConfigSettings

{

public static readonly string ConnectionString;

static ConfigSettings()

{

ConnectionString = ConfigurationManager.ConnectionStrings["DefaultConnection"].ConnectionString;

}

}

1. **Logging Initialization**: In web applications, you often need to log errors or important events. You can use a static constructor to set up your logging system once, so it’s ready to use throughout the application.

public class Logger

{

public static readonly ILogger Instance;

// Static constructor

static Logger()

{

// Initialize the logger

Instance = LogManager.GetLogger(typeof(Logger));

}

}

With this setup, you can use Logger.Instance to log messages anywhere in your application, knowing that the logger has been set up properly.

1. **Caching:** Sometimes, you need to store some data in memory so that it can be quickly accessed by different parts of your application. You can load this data into a static cache when the application starts.

public static class CacheManager

{

public static readonly Dictionary<string, string> DataCache;

// Static constructor

static CacheManager()

{

DataCache = new Dictionary<string, string>();

// Load initial data into the cache

DataCache.Add("key1", "value1");

DataCache.Add("key2", "value2");

}

}

## Calling Parametrized Constructor from Default

In C#, you can call a parameterized constructor from a default (parameterless) constructor using the this keyword. This is known as constructor chaining.

public class MyClass

{

public int Id { get; set; }

public string Name { get; set; }

// Parameterized constructor

public MyClass(int id, string name)

{

Id = id;

Name = name;

}

// Default constructor calling the parameterized constructor

public MyClass() : this(0, "Default Name")

{

// Additional initialization code (if any) can go here

}

}

**Practical Scenarios in ASP.NET**

1. **User Profiles**:

* Imagine you have a website where users can create profiles. Each profile has an ID and a name.
* Sometimes, you might create a profile with a default name and ID when the user doesn't provide these details.
* You can use a default constructor to create this default profile and a parameterized constructor to create profiles with specific details.

1. **Database Connections**:

* Think of your ASP.NET application needing to connect to a database.
* Sometimes, you might want to connect with specific settings (like a special username and password). Other times, you might use default settings.
* The default constructor can call the parameterized constructor with default settings, making it easy to manage connections.

1. **Logging Events**:

* Your application might log different events (like errors or information messages).
* For special events, you might want to include specific details. For common events, you might use default details.
* The default constructor can set up common log entries by calling the parameterized constructor with standard information.

## Inheritance in C#

**Definition**: Inheritance is a fundamental concept in object-oriented programming that allows one class (derived class) to inherit fields and methods from another class (base class). This promotes code reuse and establishes a hierarchical relationship between classes.

**Practical Scenarios**

1. **Code Reusability**: Common functionality can be defined in a base class and reused in derived classes, reducing code duplication.
2. **Polymorphism**: Inheritance supports polymorphism, allowing methods to be overridden and providing flexibility in how objects are manipulated.
3. **Extensibility**: New functionality can be added by creating new derived classes without modifying existing code, following the Open/Closed Principle.

**Diamond Problem in C#:** The Diamond Problem is a common issue in multiple inheritance scenarios where a derived class inherits from two classes that both inherit from a single base class. This can create ambiguity about which inherited property or method to use.

**Example Scenario**

Imagine we have the following class hierarchy:

1. **Base Class**: Animal
2. **Derived Classes**: Mammal and Bird both inherit from Animal
3. **Further Derived Class**: Bat inherits from both Mammal and Bird

In languages that support multiple inheritance, this can create ambiguity about which version of a method or property from Animal should be used by Bat.

Animal

/ \

Mammal Bird

\ /

\ /

Bat

**Diamond Problem Explained**

* **Ambiguity**: If both Mammal and Bird override a method from Animal, it becomes unclear which method Bat should inherit.
* **Conflict Resolution**: Some languages have complex rules to resolve these conflicts, but it can still lead to confusion and maintenance challenges.

**C# Solution: No Multiple Inheritance for Classes**

In C#, you can't inherit from more than one class directly, which prevents this problem. Instead, you can use interfaces. Interfaces are like contracts that classes agree to follow. They tell classes what methods and properties they must have, but they don't provide the actual code.

## Method Hiding

**Definition**: Method hiding allows a derived class to provide a new implementation for a method that is already defined in its base class. This is done using the new keyword. Method hiding does not override the base class method; instead, it hides it, creating a new method in the derived class.

public class BaseClass

{

public void Display()

{

Console.WriteLine("Display method in BaseClass");

}

}

public class DerivedClass : BaseClass

{

public new void Display()

{

Console.WriteLine("Display method in DerivedClass");

}

}

**Approaches to Call Hidden Methods**

1. **Using the Cast Operator**
2. **Using the base Keyword**
3. **Using a Base Class Reference**

public class BaseClass

{

public void Display()

{

Console.WriteLine("Display method in BaseClass");

}

}

public class DerivedClass : BaseClass

{

public new void Display()

{

Console.WriteLine("Display method in DerivedClass");

}

public void CallBaseDisplay()

{

base.Display(); // Calls the base class method

}

}

public class Program

{

public static void Main()

{

// Using the derived class instance

DerivedClass derivedObj = new DerivedClass();

derivedObj.Display(); // Output: Display method in DerivedClass

// Using the cast operator

((BaseClass)derivedObj).Display(); // Output: Display method in BaseClass

// Using the base keyword

derivedObj.CallBaseDisplay(); // Output: Display method in BaseClass

// Using a base class reference

BaseClass baseRefDerivedObj = new DerivedClass();

baseRefDerivedObj.Display(); // Output: Display method in BaseClass

}

}

## Polymorphism

There are two main types of polymorphism in C#: compile-time (or static) polymorphism and runtime (or dynamic) polymorphism.

**Compile-time Polymorphism**

Compile-time polymorphism is achieved through method overloading and operator overloading.

1. **Method Overloading**: This occurs when multiple methods in the same class have the same name but different parameters (type, number, or both).

public class MathOperations

{

public int Add(int a, int b)

{

return a + b;

}

public double Add(double a, double b)

{

return a + b;

}

public int Add(int a, int b, int c)

{

return a + b + c;

}

}

1. **Operator Overloading**: This allows defining custom behavior for operators in user-defined types (like classes or structs).

public class ComplexNumber

{

public int Real { get; set; }

public int Imaginary { get; set; }

public static ComplexNumber operator +(ComplexNumber c1, ComplexNumber c2)

{

return new ComplexNumber

{

Real = c1.Real + c2.Real,

Imaginary = c1.Imaginary + c2.Imaginary

};

}

}

**Runtime Polymorphism**

Runtime polymorphism is achieved through inheritance and interfaces, allowing derived classes to override methods of the base class.

1. **Method Overriding**: This allows a derived class to provide a specific implementation of a method that is already defined in its base class. This is done using the virtual keyword in the base class and the override keyword in the derived class.

public class Animal

{

public virtual void MakeSound()

{

Console.WriteLine("Animal makes a sound");

}

}

public class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("Dog barks");

}

}

public class Cat : Animal

{

public override void MakeSound()

{

Console.WriteLine("Cat meows");

}

}

With this setup, you can create an array of Animal objects and call MakeSound on each one, and the correct method for each derived class will be called:

Animal[] animals = new Animal[] { new Dog(), new Cat() };

foreach (var animal in animals)

{

animal.MakeSound(); // Output: "Dog barks" followed by "Cat meows"

}

1. **Interface Implementation**: Interfaces can be used to define methods that must be implemented by any class that implements the interface. This allows for polymorphic behavior across classes that implement the same interface.

public interface IShape

{

void Draw();

}

public class Circle : IShape

{

public void Draw()

{

Console.WriteLine("Drawing a Circle");

}

}

public class Square : IShape

{

public void Draw()

{

Console.WriteLine("Drawing a Square");

}}

Here, both Circle and Square can be treated as IShape:

IShape shape1 = new Circle();

IShape shape2 = new Square();

shape1.Draw(); // Output: "Drawing a Circle"

shape2.Draw(); // Output: "Drawing a Square"

## Properties

In C#, properties are members of a class that provide a flexible mechanism to read, write, or compute the values of private fields. Properties are a combination of methods (getters and setters) that can be used like fields, providing a clean and easy-to-use syntax for accessing and modifying data.

**Components of a Property**

A property typically has two components:

* Getter (get): Used to return the property value.
* Setter (set): Used to assign a new value to the property.

public class Person {

private string name;

// Property for 'name'

public string Name

{ get { return name; }

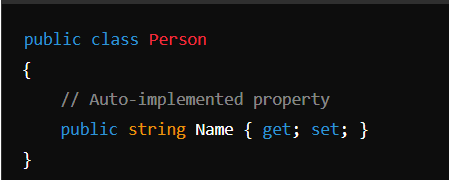
set { name = value; }

} }

In this example:

* name is a private field.
* Name is a property that allows controlled access to the name field. The get accessor returns the value of name, and the set accessor assigns a value to name.

**Auto-Implemented Properties:** C# also supports auto-implemented properties, which allow you to declare properties without defining a backing field explicitly. The compiler automatically creates a private field for you.

****

**Read-Only and Write-Only Properties**

* **Read-Only Property**: A property can be made read-only by omitting the set accessor.
  + public class Person
  + {
  + public string Name { get; } = "John Doe";
  + }
* **Write-Only Property**: A property can be made write-only by omitting the get accessor.
* public class Person
* {
* private string password;
* public string Password
* {
* set
* {
* password = value;
* }
* }
* }

## Class Vs Struct

| **Feature** | **Class** | **Struct** |
| --- | --- | --- |
| **Type** | Reference type | Value type |
| **Memory Allocation** | Allocated on the heap | Allocated on the stack or inline |
| **Inheritance** | Supports inheritance (can be derived from other classes) | Does not support inheritance (only implements interfaces) |
| **Default Constructor** | Can have a parameterless constructor | Implicit parameterless constructor (cannot define custom parameterless constructor) |
| **Immutability** | Mutable by default, can be made immutable | Often used as immutable types (using readonly fields) |
| **Copying** | Copies the reference (both variables point to the same object) | Copies the entire data (each variable has its own copy) |
| **Size** | Larger due to heap allocation and garbage collection overhead | More memory-efficient, especially for small data structures |
| **When to Use** | Use for complex data structures, polymorphism, and inheritance | Use for small, simple data structures where performance is critical |

## Interfaces

* **Definition**: An interface in C# is a contract that defines a set of methods, properties, events, or indexers that a class or struct must implement.
* **Syntax**: Interfaces are declared with the interface keyword, and implementing classes use the : symbol.
* **Multiple Implementations**: A class can implement multiple interfaces, providing flexibility and reusability in code.
* **No Implementation**: Interfaces do not provide any implementation themselves (unless using C# 8.0+ with default implementations).

**Explicit Interface Implementation**

* **Purpose**: Used when a class implements multiple interfaces that contain methods with the same signature. Explicit implementation allows the class to provide separate implementations for each interface.
* **Syntax**: The method is implemented using the InterfaceName.MethodName syntax, without an access modifier.
* **Access**: Explicitly implemented methods are only accessible through the interface reference, not directly through the class instance.

**Default Implementation in Interfaces (C# 8.0+)**

* **Definition**: Interfaces can provide default implementations for methods, allowing classes to inherit this implementation if they don't provide their own.
* **Usage**: Helps in evolving interfaces without breaking existing implementations, and reduces the need for boilerplate code in implementing classes.
* **Syntax**: Default methods are defined directly in the interface, with both the method signature and body provided.

**Handling Name Clashes with Explicit Implementation**

* **Scenario**: When a class implements two interfaces with methods having the same signature, explicit implementation is used to avoid conflicts.
* **Accessing Members**: Explicitly implemented methods can use the class's fields or properties to provide context-specific behavior for each interface.
* **Common Implementation**: If the behavior is similar across interfaces, a common private method can be used, with each interface method invoking it with specific parameters.

## Abstract Classes

Abstract classes in C# are a way to define a blueprint for other classes. They cannot be instantiated on their own and are meant to be inherited by other classes that implement their abstract members. Here's a detailed explanation:

**Key Points about Abstract Classes:**

1. **Abstract Modifier**:
   * An abstract class is declared using the abstract keyword.
   * Similarly, any method within the class that does not have an implementation is marked with the abstract keyword.
2. **Cannot be Instantiated**:
   * Abstract classes cannot be instantiated directly. That means you cannot create an object of an abstract class. Instead, you must derive a subclass from it and instantiate the subclass.
3. **Abstract Methods**:
   * An abstract method is a method that is declared in the abstract class but does not have any implementation (i.e., it is like a placeholder).
   * The derived class must override and provide an implementation for each abstract method in the abstract class.
4. **Non-Abstract Methods**:
   * An abstract class can contain methods that have full implementations. This is different from an interface, which can only have method signatures (until C# 8.0 introduced default implementations in interfaces).
5. **Inheritance**:
   * A class can inherit from an abstract class and must provide implementations for all abstract methods. If the derived class does not implement all abstract methods, it must also be declared as abstract.
6. **Use Case**:
   * Abstract classes are useful when you have a base class that should not be instantiated on its own, but instead serves as a common template for derived classes. It allows for sharing of code (through non-abstract methods) while enforcing that certain methods must be implemented by any subclass.

using System;

abstract class Animal

{

// Abstract method (no implementation here)

public abstract void MakeSound();

// Non-abstract method (with implementation)

public void Sleep()

{

Console.WriteLine("Sleeping...");

}

}

class Dog : Animal

{

// Must override the abstract method

public override void MakeSound()

{

Console.WriteLine("Woof!");

}

}

class Program

{

static void Main()

{

Dog dog = new Dog();

dog.MakeSound(); // Outputs: Woof!

dog.Sleep(); // Outputs: Sleeping...

}

}

## Delegates

**Delegates** in C# are types that represent references to methods with a specific parameter list and return type. They are used to pass methods as arguments to other methods, define callback methods, or implement event handling. Delegates are similar to function pointers in C++, but they are type-safe and secure.

**Key Features of Delegates:**

* **Type-safe**: Ensures that the signature of the method matches the signature defined by the delegate.
* **Multicast**: A delegate can reference multiple methods.
* **Object-Oriented**: Delegates are objects and can be passed as arguments, stored in variables, or returned from methods.

**Defining a Delegate**

To define a delegate, you use the delegate keyword followed by a method signature. For example:

**public delegate void MyDelegate(string message);**

This delegate can reference any method that has a void return type and takes a single string parameter.

**Using Delegates**

Once a delegate is defined, you can use it to reference methods:

1. **Declare a delegate variable**:

MyDelegate del;

1. **Assign a method to the delegate**:

del = new MyDelegate(ShowMessage);

or simply:

del = ShowMessage;

1. **Invoke the delegate**:

del("Hello, World!");

**Example of Delegates**

using System;

public delegate void MyDelegate(string message);

class Program

{

static void Main()

{

MyDelegate del = ShowMessage;

del("Hello from delegate!");

}

static void ShowMessage(string message)

{

Console.WriteLine(message);

}

}

**Multicast Delegates**

Delegates can be multicast, meaning they can reference more than one method at a time. When a multicast delegate is invoked, it calls the methods in the order they were added.

public delegate void MyDelegate(string message);

class Program

{

static void Main()

{

MyDelegate del1 = ShowMessage;

MyDelegate del2 = ShowWarning;

MyDelegate multicastDel = del1 + del2;

multicastDel("This is a multicast delegate!");

}

static void ShowMessage(string message)

{

Console.WriteLine("Message: " + message);

}

static void ShowWarning(string message)

{

Console.WriteLine("Warning: " + message);

}

}

**Delegates with Return Values**

If the delegate has a return value, only the result of the last method in the invocation list will be returned.

public delegate int MyDelegate(int x, int y);

class Program

{

static void Main()

{

MyDelegate del = Add;

del += Multiply;

int result = del(3, 4); // Result will be from Multiply method (last one in the list)

Console.WriteLine(result);

}

static int Add(int x, int y)

{

return x + y;

}

static int Multiply(int x, int y)

{

return x \* y;

}

}

**Practiced Example:**

public class Employees

{

private int \_id;

private string \_name;

private int \_salary;

private int \_experience;

public int Id { get => \_id; set => \_id = value; }

public string Name { get => \_name; set => \_name = value; }

public int Salary { get => \_salary; set => \_salary = value; }

public int Experience { get => \_experience; set => \_experience = value; }

public static void iSPromoted(List<Employees> employees, isDelegate delegateHandler)

{

foreach (Employees employee in employees) {

if (delegateHandler(employee)) {

Console.WriteLine("Employee {0} is promoted",employee.Name);

}

}

}

}

public delegate bool isDelegate(Employees employee);

public delegate int multicastDelegate(out string number);

public class MulticlassDelegates

{

public static int Number1(out string number1)

{

Console.WriteLine("First method called by Delegate");

number1 = "The value of one is : ";

return 1;

}

public static int Number2(out string number2)

{

Console.WriteLine("Second method called by Delegate");

number2 = "The value of two is : ";

return 2;

}

}

public class Delegates

{

static void Main(String[] args)

{

List<Employees> \_employees = new List<Employees>

{

new Employees { Id = 1,Name = "Tanveer",Salary = 6000,Experience = 6 },

new Employees { Id = 1,Name = "Sana",Salary = 3000,Experience = 3 },

new Employees { Id = 1,Name = "Salma",Salary = 20000,Experience = 20 },

};

//isDelegate dele = new isDelegate(Delegates.isEligibleforPromotion);

Employees.iSPromoted(\_employees, emp => emp.Salary > 5000);

//Multicast delegate example

String val = " ";

multicastDelegate ml = new multicastDelegate(MulticlassDelegates.Number1);

ml += MulticlassDelegates.Number2;

int num = ml(out val);

Console.WriteLine(val + num);

}

//public static bool isEligibleforPromotion(Employees employee)

//{

// if (employee.Salary > 5000)

// return true;

// return false;

//}

}

## Exception Handling

In C#, **exception handling** is a mechanism to handle runtime errors and ensure the program can continue or gracefully terminate. It is done using the try, catch, finally, and throw keywords. Let's go over each one in detail:

**1. try Block**

The try block contains the code that might throw an exception. It allows you to specify a block of code that you want to monitor for potential errors.

try

{

// Code that may throw an exception

int result = 10 / 0; // Division by zero will cause an exception

}

**2. catch Block**

The catch block is used to handle exceptions thrown by the try block. You can define multiple catch blocks to handle different types of exceptions.

catch (DivideByZeroException ex)

{

Console.WriteLine("Cannot divide by zero!");

}

catch (Exception ex) // General exception handler

{

Console.WriteLine("An error occurred: " + ex.Message);

}

* You can catch specific exceptions (like DivideByZeroException) or general ones (like Exception).
* The Exception class is the base class for all exceptions, so it will catch any unhandled exceptions.

**3. finally Block**

The finally block contains code that will run regardless of whether an exception was thrown or caught. It is typically used for cleanup operations (like closing files or releasing resources).

finally

{

Console.WriteLine("This will always execute.");

}

**4. throw Statement**

You can use the throw statement to manually throw an exception. This is useful if you want to raise an error condition yourself.

if (age < 0)

{

throw new ArgumentOutOfRangeException("Age cannot be negative.");

}

**Key Points:**

* **Multiple Catch Blocks**: You can catch different types of exceptions with multiple catch blocks.
* **Exception Propagation**: If an exception is not caught in a try block, it is propagated up the call stack.
* **Re-throwing Exceptions**: You can re-throw an exception using throw; in the catch block to let it propagate further.

### Inner Exceptions in C#

In C#, an **inner exception** is a property of the Exception class that holds a reference to the original exception that caused the current exception. It is used to track the sequence of exceptions that led to the final error, providing a more detailed and layered understanding of what went wrong in the application.

### Why Use Inner Exceptions?

When an exception occurs, you might catch it and then throw a new exception to provide additional context or handle the error differently. In such cases, you can pass the original exception (the one you caught) as an inner exception to the new one. This allows you to retain the original error information while adding your own message or handling.

### How to Use Inner Exceptions

When throwing a new exception, you can pass the original exception as an argument to the constructor of the new exception. This will set the InnerException property of the new exception.

#### Example

class Program

{

static void Main()

{

try

{

Method1();

}

catch (Exception ex)

{

Console.WriteLine("Caught an exception: " + ex.Message);

if (ex.InnerException != null)

{

Console.WriteLine("Inner Exception: " + ex.InnerException.Message);

}

}

}

static void Method1()

{

try

{

Method2();

}

catch (Exception ex)

{

throw new Exception("Error in Method1", ex);

}

}

static void Method2()

{

try

{

// Simulate an error

int result = int.Parse("NotANumber");

}

catch (FormatException ex)

{

throw new Exception("Error in Method2", ex);

}

}

}

## Enum

An **enum** (short for "enumeration") in C# is a value type that defines a set of named constants. It is used to represent a collection of related constants, making the code more readable and maintainable. Enums are particularly useful when you have a variable that can only take one out of a small set of possible values.

**Defining an Enum**

To define an enum, use the enum keyword followed by the name of the enum and a list of possible values enclosed in curly braces.

**Syntax:**

enum DayOfWeek

{

Sunday,

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday

}

In this example, DayOfWeek is an enum that represents the days of the week. Each value in the enum corresponds to an integer, starting from 0 by default. So, Sunday is 0, Monday is 1, and so on.

**Using Enums**

You can use enums to declare variables and assign them one of the values defined in the enum.

DayOfWeek today = DayOfWeek.Monday;

if (today == DayOfWeek.Monday)

{

Console.WriteLine("Start of the work week!");

}

**Underlying Type**

By default, the underlying type of each enum member is int, but you can specify a different underlying type by using a colon followed by the type after the enum name.

enum DayOfWeek : byte

{

Sunday,

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday

}

In this example, the underlying type of DayOfWeek is byte, so each value will be of type byte instead of int.

**Assigning Specific Values**

You can assign specific values to the members of an enum. If you don't specify a value for a member, it will take the next available integer value.

enum DayOfWeek

{

Sunday = 1,

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday

}

In this case, Sunday is 1, Monday is 2, and so on.

**Enum Methods**

Enums have some useful methods that can help you work with them:

* **ToString()**: Converts the enum value to its name as a string.

DayOfWeek today = DayOfWeek.Monday;

Console.WriteLine(today.ToString()); // Outputs: "Monday"

* **Enum.Parse()**: Converts a string to the corresponding enum value.

DayOfWeek day = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Friday");

* **Enum.GetValues()**: Returns an array of the values in the enum.

foreach (DayOfWeek day in Enum.GetValues(typeof(DayOfWeek)))

{

Console.WriteLine(day);

}

* **Enum.GetNames()**: Returns an array of the names in the enum.

foreach (string name in Enum.GetNames(typeof(DayOfWeek)))

{

Console.WriteLine(name);

}

## Regions

In C#, **regions** are used to organize and collapse blocks of code in your Visual Studio code editor, making it easier to navigate large code files. Regions do not affect the functionality of your code but serve as a tool for improving code readability and maintainability.

**Syntax**

A region in C# is defined using #region and #endregion directives:

#region RegionName

// Your code here

#endregion

## Access Modifiers

| **Element** | **Public** | **Private** | **Protected** | **Internal** | **Protected Internal** | **Private Protected** | **Default Access Modifier** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Type** | Accessible from anywhere | N/A | N/A | Accessible within the same assembly | N/A | N/A | Internal |
| **Type Field** | Accessible from anywhere | Accessible only within the type | Accessible within the type and derived types | Accessible within the same assembly | Accessible within the same assembly or derived types | Accessible within the type and derived types in the same assembly | Private |

## Attributes

* Attributes allow you to add declarative information to your programs. This information can then be queried at runtime using reflection.
* **A few pre-defined attributes with in the .NET framework.**  
  Obsolete - Marks types and type members outdated  
  WebMethod - To expose a method as an XML Web service method  
  Serializable - Indicates that a class can be serialized
* **Example program using pre defined Obsolete attribute:**  
  Obsolete attribute can be used with types or type members that are **obsolete (Outdated).** If a developer uses a type or a type member that is decorated with obsolete attribute, the compiler issues a warning or an error depending on how the attribute is configured.

In this sample program, **Add(int FirstNumber, int SecondNumber)** method is decorated with [Obsolete] attribute. If you compile this program, in the output window you will see a warning message (Compile complete -- 0 errors, 1 warnings). Also, visual studio, shows a green squiggly line under the  **Add(int FirstNumber, int SecondNumber)** method. If you hover the mouse over the squiggly line, you should see the warning message.  
  
using System;  
using System.Collections.Generic;  
public class MainClass  
{  
    private static void Main()  
    {  
        Calculator.Add(10, 15);  
    }  
}  
public class Calculator  
{  
    [Obsolete]  
    public static int Add(int FirstNumber, int SecondNumber)  
    {  
        return FirstNumber + SecondNumber;  
    }  
    public static int Add(List<int> Numbers)  
    {  
        int Sum = 0;  
        foreach (int Number in Numbers)  
        {  
            Sum = Sum + Number;  
        }  
        return Sum;  
    }  
}

The warning message says **'Calculator.Add(int, int)' is obsolete**. However, this message is not completely useful, because it says**'Calculator.Add(int, int)' is obsolete**, but not tell us which other method **should we be using instead**. So this is when we can customize, the warning message**using attribute parameters**.

The intention of the developer of Calculator class is that, he wanted us to use **Add(List<int> Numbers)**, instead of int **Add(int FirstNumber, int SecondNumber)**. To communicate this message we can customize the warning message using attribute parameters as shown below. With this customization we are not only communicating that **Add(int FirstNumber, int SecondNumber)** method is obsolete, we are also telling to use the alternative method that is available.

[Obsolete("Use Add(List<int> Numbers) instead")]  
public static int Add(int FirstNumber, int SecondNumber)  
  
If you want to generate a compiler error instead of warning, pass true for the bool error parameter of the Obsolete attribute as shown below. Now, we can't even compile the program.  
[Obsolete("Use Add(List<int> Numbers) instead", true)]  
public static int Add(int FirstNumber, int SecondNumber)  
  
Finally, If you right click on Obsolete attribute and select Go To Definition, you will see that, an attribute is nothing but a class that inherits from System.Attribute base class.

## Reflection

**Reflection** is the ability of inspecting an assemblie's metadata at runtime.  It is used to find all types in an assembly and/or dynamically invoke methods in an assembly. This includes information about the type, properties, methods, and events of an object. With Reflection, we can dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.There are several uses of reflection.

Some Practical Applications:

**1.** When you drag and drop a button on a win forms or an asp.net application. The properties window uses reflection to show all the properties of the Button class. So,reflection is extensivley used by IDE or a UI designers.  
**2.** Late binding can be achieved by using reflection. You can use reflection to dynamically create an instance of a type, about which we don't have any information at compile time. So, reflection enables you to use code that is not available at compile time.  
**3.** Consider an example where we have two alternate implementations of an interface. You want to allow the user to pick one or the other using a config file. With reflection, you can simply read the name of the class whose implementation you want to use from the config file, and instantiate an instance of that class. This is another example for late binding using reflection.

**So, in short reflection can be used for type discovery (i.e finding methods, properties, events, fields, constructors etc) and late binding.**

**Example:**

using System;  
using System.Reflection;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            // Get the Type Using GetType() static method  
            Type T = Type.GetType("Pragim.Customer");  
            // Print the Type details  
            Console.WriteLine("Full Name = {0}",T.FullName);  
            Console.WriteLine("Just the Class Name = {0}",T.Name);  
            Console.WriteLine("Just the Namespace = {0}", T.Namespace);  
            Console.WriteLine();  
            // Print the list of Methods  
            Console.WriteLine("Methods in Customer Class");  
            MethodInfo[] methods = T.GetMethods();  
            foreach (MethodInfo method in methods)  
            {  
                // Print the Return type and the name of the method  
                Console.WriteLine(method.ReturnType.Name + " " + method.Name);  
            }  
            Console.WriteLine();  
            //  Print the Properties  
            Console.WriteLine("Properties in Customer Class");  
            PropertyInfo[] properties = T.GetProperties();  
            foreach (PropertyInfo property in properties)  
            {  
                // Print the property type and the name of the property  
                Console.WriteLine(property.PropertyType.Name + " " + property.Name);  
            }  
            Console.WriteLine();  
            //  Print the Constructors  
            Console.WriteLine("Constructors in Customer Class");  
            ConstructorInfo[] constructors = T.GetConstructors();

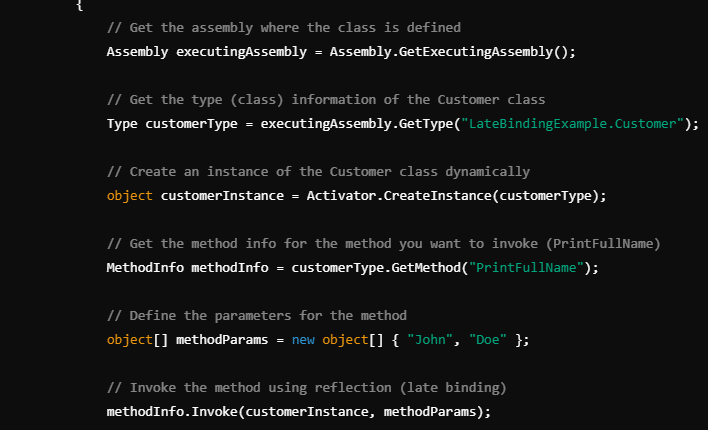
            foreach (ConstructorInfo constructor in constructors)  
            {  
                Console.WriteLine(constructor.ToString());  
            }  
        }  
    }  
    public class Customer  
    {  
        public int Id { get; set; }  
        public string Name { get; set; }  
  
        public Customer(int ID, string Name)  
        {  
            this.Id = ID;  
            this.Name = Name;  
        }  
  
        public Customer()  
        {  
            this.Id = -1;  
            this.Name = string.Empty;  
        }

        public void PrintID()  
        {  
            Console.WriteLine("ID = {0}", this.Id);  
        }  
        public void PrintName()  
        {  
            Console.WriteLine("Name = {0}", this.Name);  
        }  
    }

* In this example to get the type of customer class we have used **GetType()** static method defined on the **Type**class. We pass in the fully qualified name of the type including the namespace as a parameter to the GetType() method.  
  Type T = Type.GetType("Pragim.Customer");
* **To get the type information we have the following 2 ways as well.**  
  **Use typeof keyowrd**  
  Type T = typeof(Customer);
* **Use GetType() on the instance of the customer class.**  
  Customer C1 = new Customer();  
  Type T = C1.GetType();
* To get the methods information, we use Type.GetMethods(), which returns MethodInfo[] array and along the same lines we use Type.GetProperties() to get properties information, but Type.GetProperties() returns PropertyInfo[] array.

**Late Binding**

* **Late binding** in C# refers to the process where method calls or property access are resolved at **runtime** rather than at compile-time. This is in contrast to **early binding**, where the compiler resolves method calls and property accesses during compilation.
* In late binding, the types, methods, or properties being accessed are not known at compile-time, and the actual method or property that gets invoked is determined dynamically at runtime. This is typically done using reflection or interfaces like IDispatch in COM.
* **How Late Binding Works:**
* Late binding is usually achieved using the Reflection API in C#.
* You retrieve information about the types, methods, and properties of objects dynamically at runtime.
* You invoke methods or access properties without needing to explicitly know about the class at compile-time.
* **Use Case of Late Binding:**
* Late binding is useful in scenarios where:
* You don't know the types you are going to use until runtime.
* You need to interact with objects from other assemblies or components that may change frequently.
* You're working with libraries like COM (Component Object Model) or plugins, where objects can change dynamically.
* **Example of Late Binding:**

****

* **Key Steps in Late Binding:**

1. **Assembly Loading**: Load the assembly that contains the class.
2. **Type Discovery**: Get the type of the class from the assembly.
3. **Instance Creation**: Create an instance of the class using Activator.CreateInstance().
4. **Method/Property Discovery**: Get the method or property information using MethodInfo or PropertyInfo.
5. **Method Invocation**: Use the MethodInfo.Invoke() to invoke the method on the class instance.

* **Advantages of Late Binding:**
* **Flexibility**: Late binding allows for more flexible code, as it can work with objects and types not known until runtime.
* **Dynamic Scenarios**: It is useful when integrating with dynamically changing environments like plugins, COM objects, or external libraries.
* **Loose Coupling**: It helps in scenarios where tight coupling between assemblies is undesirable.
* **Disadvantages of Late Binding:**
* **Performance Overhead**: Late binding is slower compared to early binding since the method resolution happens at runtime.
* **Lack of Compile-Time Safety**: Since the method resolution happens at runtime, the compiler cannot check for errors, making it easier to run into runtime exceptions if a method or property does not exist.
* **Complexity**: It adds complexity to the code and can make it harder to debug.
* **When to Use Late Binding:**
* When you need to load types dynamically at runtime, such as loading plugins or modules.
* When dealing with COM components or external libraries that may change or are unknown at compile time.
* When interacting with a scripting environment where types may not be known in advance.

## Generics

using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            bool Equal = Calculator.AreEqual(1, 2);  
            if (Equal)  
            {  
                Console.WriteLine("Equal");  
            }  
            else  
            {  
                Console.WriteLine("Not Equal");  
            }  
        }  
    }  
    public class Calculator  
    {  
        public static bool AreEqual(int value1, int value2)  
        {  
            return value1 == value2;  
        }  
    }  
}  
**It's a compile time error to invoke AreEqual() method with string parameters.**  
bool Equal = Calculator.AreEqual("A", "B");  
One way of making AreEqual() method reusable, is to use **object**type parameters. Since, every type in .NET directly or indirectly inherit from **System.Object**type, AreEqual() method works with any data type, but the problem is performance degradation due to boxing and unboxing happening.   
Also, AreEuqal() method is no longer type safe. It is now possible to pass integer for the first parameter, and a string for the second parameter. It doesn't really make sense to compare strings with integers.   
using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            bool Equal = Calculator.AreEqual("A", "B");  
            if (Equal)  
            {  
                Console.WriteLine("Equal");  
            }  
            else  
            {  
                Console.WriteLine("Not Equal");  
            }  
        }  
    }  
    public class Calculator  
    {  
        public static bool AreEqual(object value1, object value2)  
        {  
            return value1 == value2;  
        }  
    }  
}  
**So, the probem with using System.Object type is that**  
1. AreEqual() method is not type safe  
2. Performance degradation due to boxing and unboxing.  
Both of these issues can be solved with generics and still make AreEqual() method work with different data types. The re written example using generics is shown below.   
using System;  
namespace Pragim  
{  
    public class MainClass  
    {  
        private static void Main()  
        {  
            bool Equal = Calculator.AreEqual<int>(2, 1);  
            if (Equal)  
            {  
                Console.WriteLine("Equal");  
            }  
            else  
            {  
                Console.WriteLine("Not Equal");  
            }  
        }  
    }  
    public class Calculator  
    {  
        public static bool AreEqual<T>(T value1, T value2)  
        {  
            return value1.Equals(value2);  
        }  
    }  
}  
To make AreEqual() method generic, we specify a type parameter using angular brackets as shown below.  
public static bool AreEqual<T>(T value1, T value2)  
At the point, When the client code wants to invoke this method, they need to specify the type, they want the method to operate on. If the user wants the AreEqual() method to work with integers, they can invoke the method specifying int as the datatype using angular brackets as shown below.  
bool Equal = Calculator.AreEqual<int>(2, 1);  
To operate with string data type  
bool Equal = Calculator.AreEqual<string>("A", "B");  
In this example, we made the method generic. Along the same lines, it is also possible to make classes, interfaces and delegates generic.

## Overriding ToString() and Equal() methods

In C#, ToString and Equals are methods that can be overridden to provide custom behavior for your classes. Here's how they work and why you might override them:

**1. Overriding ToString() Method**

The ToString() method is inherited from the Object class, and it's used to return a string that represents the current object. By default, it returns the full name of the class (namespace + class name), but you often override it to provide a meaningful representation of the object.

**Example:**

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

// Override the ToString method

public override string ToString()

{

return $"{FirstName} {LastName}";

}

}

Person person = new Person { FirstName = "John", LastName = "Doe" };

Console.WriteLine(person.ToString()); // Outputs: John Doe

**Why Override ToString()?**

* To give a more useful or human-readable string representation of an object.
* Helpful for debugging and logging, so you can quickly see the state of an object.

**2. Overriding Equals() Method**

The Equals() method is also inherited from Object. By default, it performs a reference comparison, meaning it checks if two objects refer to the same memory location. However, when you override it, you can define equality based on the object's properties (i.e., value-based comparison).

**Example:**

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

// Override Equals method for value-based comparison

public override bool Equals(object obj)

{

if (obj == null || GetType() != obj.GetType())

return false;

Person other = (Person)obj;

return FirstName == other.FirstName && LastName == other.LastName;

}

// Always override GetHashCode when overriding Equals

public override int GetHashCode()

{

return HashCode.Combine(FirstName, LastName);

}

}

Person person1 = new Person { FirstName = "John", LastName = "Doe" };

Person person2 = new Person { FirstName = "John", LastName = "Doe" };

Console.WriteLine(person1.Equals(person2)); // Outputs: True

**Why Override Equals()?**

* To define object equality based on property values rather than memory location.
* Useful for comparisons between objects in collections, or when checking if two objects represent the same data.

**Important: Always Override GetHashCode() When You Override Equals()**

The GetHashCode() method should be overridden along with Equals(), especially if your object will be used as a key in collections like Dictionary or HashSet. This ensures that objects considered equal produce the same hash code.

## Convert.ToString() VS ToString

In C#, both Convert.ToString() and ToString() methods are used to convert an object to its string representation. However, they differ in their behavior and usage. Let's explore the key differences between them:

**1. ToString() Method**

* **Purpose:** This method is called directly on an object to get its string representation.
* **Usage:** It is used when you know the object is not null because it will throw a NullReferenceException if you attempt to call ToString() on a null object.
* **Custom Implementation:** You can override the ToString() method in your own classes to provide a custom string representation.

**Example:**

object obj = 123;

Console.WriteLine(obj.ToString()); // Outputs: "123"

string nullString = null;

// Console.WriteLine(nullString.ToString()); // Throws NullReferenceException

**Key Points:**

* The ToString() method is available on all objects because it's inherited from the System.Object class.
* It throws an exception if the object is null.

**2. Convert.ToString() Method**

* **Purpose:** This method is part of the Convert class and is designed to safely convert an object to a string, even if the object is null.
* **Usage:** Use Convert.ToString() when you're not sure if the object might be null, as it handles null gracefully by returning an empty string ("").
* **Doesn't Need Overriding:** This method does not depend on any custom implementation of ToString(), but it internally calls ToString() on non-null objects.

**Example:**

object obj = 123;

Console.WriteLine(Convert.ToString(obj)); // Outputs: "123"

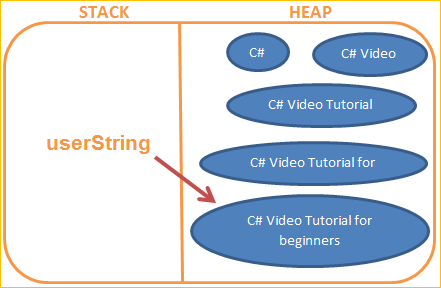
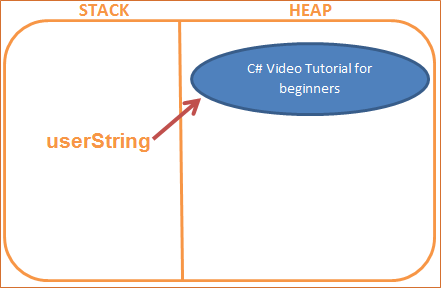
string nullString = null;

Console.WriteLine(Convert.ToString(nullString)); // Outputs: ""

**Key Points:**

* It safely converts null objects into an empty string, preventing exceptions.
* Ideal for scenarios where you don't want to worry about null reference issues.

## String Vs String Builder

**Let's understand the meaning of mutable and immutable strings with an example.**  
using System;  
public class MainClass  
{  
    public static void Main()  
    {  
        string userString = "C#";  
        userString += " Video";  
        userString += " Tutorial";  
        userString += " for";  
        userString += " beginners";  
        Console.WriteLine(userString);  
    }  
}  
**In this example, userString variable is changed 5 times.**  
1. C#  
2. C# => C# Video  
3. C# Video => C# Video Tutorial  
4. C# Video Tutorial => C# Video Tutorial for  
5. C# Video Tutorial for => C# Video Tutorial for beginners  
  
Since, userString variable is of type System.String, and when we change this string 5 times, we end up with 5 string objects on the heap as shown in the diagram below. Immutable means, once a string object is created it cannot be changed, without creating another new string object. So in our example, When we initialize userString variable to **"C#"** we get one immutable string object on the heap. When we concatenate **" Video"** word to userString variable, the first created**"C#"**string object is orphaned(userString variable no longer points to this object). Now another new string object with words **"C# Video"** will be created to which the userString variable points to. So this process continues until, userString reference variable, points to the last string object (**C# Video Tutorial for beginners**), leaving the other 4 string onbjects on the heap(orphaned), until they are garbage collected, increasing the pressure on memory.  
  
  
But on the other hand, StringBuilder string objects are mutable, meaning they can be changed inplace, without the need of creating another new StringBuilder object. The above example is rewritten using StringBuilder object.  
using System;  
using System.Text;  
namespace Pragim  
{  
    public class MainClass  
    {  
        public static void Main()  
        {  
            StringBuilder userStringBuilder =   
                new StringBuilder("C#");  
            userStringBuilder.Append(" Video");  
            userStringBuilder.Append(" Tutorial");  
            userStringBuilder.Append(" for");  
            userStringBuilder.Append(" beginners");  
            Console.WriteLine(userStringBuilder.ToString());  
        }  
    }  
}  
With StringBuilder, no matter how many times you manipulate a string, you will ever have only one instance.   
  
  
**So in brief, here are the differences between String and StringBuilderobjects.**  
1. Objects of type StringBuilder are mutable where as objects of type System.String are immutable.   
2. As StringBuilder objects are mutable, they offer better performance than string objects of type System.String.  
3. StringBuilder class is present in System.Text namespace where String class is present in System namespace.

## Partial Classes

**Partial classes allow us to split a class into 2 or more files.**  All these parts are then combined into a single class, when the application is compiled. The partial keyword can also be used to split a struct or an interface over two or more files.  
**Let us split this class into 2 files.**One file is going to contain, the private fields and public properties, and the other file is going to contain the public method. Right click on the web application project, and add a class file, with name **PartialCustomerOne.cs**. Notice, that the **PartialCustomer** class is marked with the **partial** keyword and it contains, only, the 2 private fields and the public properties.   
public partial class PartialCustomer  
{  
    private string \_firstName;  
    private string \_lastName;  
  
    public string FirstName  
    {  
        get { return \_firstName; }  
        set { \_firstName = value; }  
    }  
  
    public string LastName  
    {  
        get { return \_lastName; }  
        set { \_lastName = value; }  
    }  
}  
**Now, add another class file with name, PartialCustomerTwo.cs**. Notice that, the **PartialCustomer** class, in this file is also marked as a **partial** class, and contains only the public method - **GetFullName()**. We are able to access the private fields, **\_firstName** and **\_lastName**, that are defined in **PartialCustomerOne.cs** file.  
public partial class PartialCustomer  
{  
    public string GetFullName()  
    {  
        return \_firstName + ", " + \_lastName;  
    }  
}  
**Copy and paste the following code in the Page\_Load() event of the webform1.** Though, the **PartialCustomer** class is split across 2 files(PartialCustomerOne.cs and PartialCustomerTwo.cs), we are able to use it the same way as the Customer class.  
Customer c1 = new Customer();  
c1.FirstName = "Pragim";  
c1.LastName = "Technologies";  
string FullName1 = c1.GetFullName();  
Response.Write("Full Name = " + FullName1 + "<br/>");  
PartialCustomer c2 = new PartialCustomer();  
c2.FirstName = "Pragim";  
c2.LastName = "Tech";  
  
string FullName2 = c2.GetFullName();  
Response.Write("Full Name = " + FullName2 + "<br/>");  
  
**Advantages of partial classes**  
**1.** The main advantage is that, visual studio uses partial classes to separate, automatically generated system code from the developer's code. For example, when you add a webform, two .CS files are generated  
**a) WebForm1.aspx.cs -**Contains the developer code  
**b) WebForm1.aspx.designer.cs -** Contains the system generated code. For example, declarations for the controls that you drag and drop on the webform.  
**2.** When working on large projects, spreading a class over separate files allows multiple programmers to work on it simultaneously. Though, microsoft claims this as an advantage, I haven't really seen anywhere, people using partial classes, just to work on them simultaneously.