# 01\_Intro

### **1. Basic Structure of a C# Program**

// Namespace Declaration

using System;

class Pragim

{

public static void Main()

{

// Write to console

Console.WriteLine("Welcome to PRAGIM Technologies!");

}

}

#### **a. Namespace Declaration (using System;)**

* **Purpose**: The line using System; indicates that your program is using the System namespace, which is a collection of classes, interfaces, and other types provided by the .NET framework. The System namespace includes many essential classes like Console, String, DateTime, etc.
* **Without using System;**: If you omit this line, you would need to fully qualify the classes you use. For example, instead of writing Console.WriteLine("Welcome to PRAGIM Technologies!");, you would have to write System.Console.WriteLine("Welcome to PRAGIM Technologies!");.

#### **b. Class Declaration (class Pragim)**

* **Purpose**: class Pragim defines a class named Pragim. In C#, a class is a blueprint for creating objects. It can contain methods, fields, properties, and other members. The Pragim class in this example contains one method: Main.

#### 

#### **c. Main Method (public static void Main())**

* **Purpose**: The Main method is the entry point of a C# application. When you run the program, the execution starts from this method.
  + **public**: The Main method is accessible from outside the class. This is necessary because the runtime needs to access it to start the program.
  + **static**: The Main method is static, meaning it belongs to the class itself rather than an instance of the class. This allows the runtime to invoke the Main method without creating an instance of the class.
  + **void**: The Main method doesn’t return any value.
  + **Main()**: This is the method signature. The parentheses indicate that the method takes no parameters in this case.

#### **d. Console.WriteLine()**

* **Purpose**: Console.WriteLine("Welcome to PRAGIM Technologies!"); is a statement that writes the specified string to the console window.
  + **Console**: This is a class provided by the System namespace, and it’s used for interacting with the console (standard input, output, and error streams).
  + **WriteLine**: This method of the Console class writes a line of text to the console.

### **2. Purpose of using System;**

* **Explanation**: As mentioned above, the using System; directive allows you to use the classes and methods defined in the System namespace without having to prefix them with System. each time. It's a way to simplify your code and make it more readable.

### **3. Purpose of Main() Method**

* **Explanation**: The Main method serves as the entry point of your C# application. When you execute your program, the .NET runtime looks for this method to start the application. Without a Main method, your program wouldn’t know where to begin execution, which is why it’s required in every C# console application.

# 02\_Input/Output

using System;

class Program

{

static void Main()

{

// Prompt the user for his name

Console.WriteLine("Please enter your name");

// Read the name from console

string UserName = Console.ReadLine();

// Concatenate name with hello word and print

Console.WriteLine("Hello " + UserName);

// Placeholder syntax to print name with hello word

//Console.WriteLine("Hello {0}", UserName);

}

}

### **1. Reading from the Console**

* **Purpose**: Reading from the console allows your program to take input from the user.

**Console.ReadLine()**: This method reads an entire line of text input from the console until the user presses Enter. It returns the input as a string.  
  
string UserName = Console.ReadLine();

* In this example, whatever the user types into the console is stored in the UserName variable.

### **2. Writing to the Console**

* **Purpose**: Writing to the console is how your program can output information to the user.

**Console.WriteLine()**: This method writes the specified data, followed by a newline character, to the console.  
  
Console.WriteLine("Please enter your name");

* This line outputs the text "Please enter your name" to the console.

### **3. Two Ways to Write to the Console**

#### **a. Concatenation**

**Explanation**: Concatenation involves joining strings together using the + operator. It’s a straightforward method to combine multiple pieces of data into a single string.  
  
Console.WriteLine("Hello " + UserName);

* Here, the string "Hello " is concatenated with the value stored in UserName. If the user entered "John," the output would be "Hello John".

#### **b. Placeholder Syntax (Preferred Method)**

**Explanation**: Placeholder syntax is a more flexible and preferred way to format strings in C#. It allows you to embed variables within a string in a clear and organized manner. The placeholders are indicated by {index}, where index is the position of the variable in the list of arguments provided to Console.WriteLine.  
  
Console.WriteLine("Hello {0}", UserName);

* In this example, {0} is a placeholder that will be replaced by the value of UserName. If UserName is "John," the output will again be "Hello John". This method is preferred because it makes the code cleaner and easier to maintain, especially when you need to include multiple variables.

### **4. Case Sensitivity in C#**

* **Explanation**: C# is a case-sensitive language, meaning that it treats uppercase and lowercase letters as different. For example, Console.WriteLine is not the same as console.writeline. It’s important to consistently use the correct case for identifiers like variable names, methods, and class names to avoid errors.

using System;

class Program

{

static void Main()

{

// Prompt the user to enter first name, last name, and age in one line

Console.WriteLine("Please enter your first name, last name, and age separated by spaces:");

string input = Console.ReadLine();

// Split the input string into parts

string[] inputs = input.Split(' ');

// Assign each part to a variable

string firstName = inputs[0];

string lastName = inputs[1];

string age = inputs[2];

// Output the gathered information

Console.WriteLine("Hello {0} {1}, you are {2} years old.", firstName, lastName, age);

}

}

This approach can be useful when you expect the user to input multiple pieces of information in a single line.

The Split(' ') method splits the input string into an array of strings based on spaces.

The resulting array inputs contains the first name, last name, and age as separate elements.

# 03\_Data Types

## Built-in Types in C#

C# provides several built-in types that are essential for handling different kinds of data. Here’s an overview of each category:

#### **1. Boolean Type**

* **Type**: bool
* **Values**: true or false

**Usage**: Boolean values are used for logical conditions and control flow in your programs.  
  
bool isStudent = true;

#### **2. Integral Types**

* **Signed Types**: Can store both positive and negative values.
  + sbyte: 8-bit signed integer (-128 to 127)
  + short: 16-bit signed integer (-32,768 to 32,767)
  + int: 32-bit signed integer (-2,147,483,648 to 2,147,483,647)
  + long: 64-bit signed integer (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
* **Unsigned Types**: Can store only non-negative values.
  + byte: 8-bit unsigned integer (0 to 255)
  + ushort: 16-bit unsigned integer (0 to 65,535)
  + uint: 32-bit unsigned integer (0 to 4,294,967,295)
  + ulong: 64-bit unsigned integer (0 to 18,446,744,073,709,551,615)
* **Character Type**:
  + char: Represents a single Unicode character (e.g., 'A', '9').

int age = 25;

char initial = 'A';

#### **3. Floating Types**

* **Types**:
  + float: 32-bit floating point (approximately 6-7 decimal digits of precision).
  + double: 64-bit floating point (approximately 15-16 decimal digits of precision).

float price = 99.99f;

double distance = 12345.6789;

#### **4. Decimal Type**

* **Type**: decimal

**Usage**: Ideal for financial and monetary calculations where precision is critical. It’s a 128-bit data type with 28-29 significant digits of precision.  
  
decimal salary = 10000.50m;

#### **5. String Type**

* **Type**: string

**Usage**: Represents a sequence of characters. Strings are immutable, meaning that once created, they cannot be changed.  
  
string name = "John Doe";

### 

### 

### **Escape Sequences in C#**

Escape sequences are special characters in a string that represent certain characters which might otherwise be hard to include directly. Some common escape sequences are:

* \n: Newline
* \t: Tab
* \\: Backslash
* \": Double quote

For example:

string path = "C:\\Pragim\\DotNet\\Training\\Csharp";

In this case, \\ is used to represent a single backslash.

### **Verbatim Literals**

A verbatim literal string is prefixed with an @ symbol. It treats escape sequences as regular characters, making file paths and strings that contain backslashes more readable.

**Without Verbatim Literal**:  
  
string path = "C:\\Pragim\\DotNet\\Training\\Csharp";

**With Verbatim Literal**:  
  
string path = @"C:\Pragim\DotNet\Training\Csharp";

Using @ in front of a string also allows multi-line strings:

string multiLine = @"This is a

multi-line string

using a verbatim literal.";

### **Practical Examples**

using System;

class Program

{

static void Main()

{

// Boolean type example

bool isActive = true;

// Integral types example

int age = 30;

char grade = 'A';

// Floating point types example

float pi = 3.14f;

double e = 2.71828;

// Decimal type example

decimal accountBalance = 1500.75m;

// String type example

string name = "Alice";

// Using escape sequences

string path = "C:\\Pragim\\DotNet\\Training\\Csharp";

Console.WriteLine("Path using escape sequences: " + path);

// Output: Path using escape sequences: C:\Pragim\DotNet\Training\Csharp

// Using verbatim literal

string verbatimPath = @"C:\Pragim\DotNet\Training\Csharp";

Console.WriteLine("Path using verbatim literal: " + verbatimPath);

// Output: Path using verbatim literal: C:\Pragim\DotNet\Training\Csharp

}

}

## String Data Type

using System;

namespace ConsoleApplication1

{

class Program

{

public static void Main()

{

// Displaying double quotes in C#

string Name = "\"Pragim\"";

Console.WriteLine(Name);

// Expected Output: "Pragim"

// Displaying new line character in C#

Name = "One\nTwo\nThree";

Console.WriteLine(Name);

// Expected Output:

// One

// Two

// Three

// Displaying backslashes in a path using escape sequences

Name = "c:\\Pragim\\DotNet\\Training\\Csharp";

Console.WriteLine(Name);

// Expected Output: c:\Pragim\DotNet\Training\Csharp

// C# verbatim literal for file path

Name = @"c:\Pragim\DotNet\Training\Csharp";

Console.WriteLine(Name);

// Expected Output: c:\Pragim\DotNet\Training\Csharp

}

}

# 04\_Operators

### **Common Operators in C#**

1. **Assignment Operator (=)**: Assigns a value to a variable.
2. **Arithmetic Operators (+, -, \*, /, %)**: Perform basic arithmetic operations.
3. **Comparison Operators (==, !=, >, >=, <, <=)**: Compare values and return a boolean result.
4. **Conditional Operators (&&, ||)**: Used to perform logical operations.
5. **Ternary Operator (?:)**: A shorthand for if-else statements.
6. **Null Coalescing Operator (??)**: Returns the left-hand operand if it is not null; otherwise, it returns the right-hand operand.

### **Examples from the Demo**

#### **Example 1: Assignment and Arithmetic Operators**

using System;

namespace ConsoleApplication1

{

class Program

{

public static void Main()

{

// Assignment Operator example

int i = 10; // Assigning 10 to variable i

bool b = true; // Assigning true to variable b

// For dividing 2 numbers we can use either

// % or / operators

int numerator = 10;

int denominator = 2;

// Arithmetic operator / returns quotient

int quotient = numerator / denominator;

Console.WriteLine("Quotient = {0}", quotient);

// Expected Output: Quotient = 5

// Arithmetic operator % returns remainder

int remainder = numerator % denominator;

Console.WriteLine("Remainder = {0}", remainder);

// Expected Output: Remainder = 0

}

}

}

* **Assignment Operator (=)**: Assigns the value 10 to i and true to b.
* **Arithmetic Operators (/, %)**:
  + / returns the quotient (5).
  + % returns the remainder (0).

#### **Example 2: Comparison Operators**

using System;

namespace ConsoleApplication1

{

class Program

{

public static void Main()

{

int number = 10;

// To compare if 2 numbers are equal use comparison operator ==

if (number == 10)

{

Console.WriteLine("Number is equal to 10");

// Expected Output: Number is equal to 10

}

// To compare if 2 numbers are not equal use comparison operator !=

if (number != 5)

{

Console.WriteLine("Number is not equal to 5");

// Expected Output: Number is not equal to 5

}

}

* **Comparison Operator (==)**: Checks if number is equal to 10.
* **Comparison Operator (!=)**: Checks if number is not equal to 5.

#### **Example 3: Conditional Operators**

using System;

namespace ConsoleApplication1

{

class Program

{

public static void Main()

{

int number1 = 10;

int number2 = 20;

// && operator: Both conditions must be true

if (number1 == 10 && number2 == 20)

{

Console.WriteLine("Both conditions are true");

// Expected Output: Both conditions are true

}

// || operator: At least one condition must be true

number2 = 21; // Changing number2 to 21

if (number1 == 10 || number2 == 20)

{

Console.WriteLine("At least one of the conditions is true");

// Expected Output: At least one of the conditions is true

}

}

}

}

* **Conditional Operator (&&)**: Both conditions (number1 == 10 and number2 == 20) must be true.
* **Conditional Operator (||)**: At least one condition (number1 == 10 or number2 == 20) must be true.

#### **Example 4: Ternary Operator**

Without Ternary Operator:

using System;

namespace ConsoleApplication1

{

class Program

{

public static void Main()

{

int number = 10;

bool isNumber10;

if (number == 10)

{

isNumber10 = true;

}

else

{

isNumber10 = false;

}

Console.WriteLine("Number == 10 is {0}", isNumber10);

// Expected Output: Number == 10 is True

}

}

}

With Ternary Operator:

using System;

namespace ConsoleApplication1

{

class Program

{

public static void Main()

{

int number = 10;

// Ternary operator example

bool isNumber10 = number == 10 ? true : false;

Console.WriteLine("Number == 10 is {0}", isNumber10);

// Expected Output: Number == 10 is True

}

}

}

* **Ternary Operator (?:)**: A more concise way to write if-else. It checks if number == 10 and assigns true or false to isNumber10 accordingly.

#### **Null Coalescing Operator Example**

using System;

namespace ConsoleApplication1

{

class Program

{

public static void Main()

{

string name = null;

string defaultName = "Default Name";

// Null Coalescing Operator

string finalName = name ?? defaultName;

Console.WriteLine("Name: " + finalName);

// Expected Output: Name: Default Name

}

}

}

* **Null Coalescing Operator (??)**: Returns defaultName if name is null.

# 05\_Nullable Data Type

### **Understanding Nullable Types in C#**

In C#, types are generally divided into two broad categories:

1. **Value Types**: These include int, float, double, structs, enums, etc.
2. **Reference Types**: These include interface, class, delegates, arrays, etc.

By default, **value types** are non-nullable, meaning they cannot hold a null value. However, in many cases, particularly when dealing with databases, we may want to represent the absence of a value using null. This is where **nullable types** come into play.

### **Making Value Types Nullable**

To make a value type nullable, you use the ? symbol after the type. This allows the value type to hold either a value of its type or null.

#### **Examples:**

int i = 0; // i is non-nullable, cannot be set to null

int? j = null; // j is nullable, can be set to null

* int i = 0; is a non-nullable integer. Trying to set i = null; would result in a compiler error.
* int? j = null; is a nullable integer, so setting j = null; is perfectly valid.

### **Nullable Types and Database Compatibility**

Nullable types are particularly useful when working with databases, where fields might not have values. Nullable types bridge the gap between the non-nullable value types in C# and the potentially null fields in a database.

### **Example Program Without Using the Null Coalescing Operator**

Here’s a basic program that checks if a value is null and handles it accordingly:

using System;

class Program

{

static void Main()

{

int AvailableTickets;

int? TicketsOnSale = null;

if (TicketsOnSale == null)

{

AvailableTickets = 0; // Assign 0 if TicketsOnSale is null

}

else

{

AvailableTickets = (int)TicketsOnSale; // Cast and assign if not null

}

Console.WriteLine("Available Tickets={0}", AvailableTickets);

// Expected Output: Available Tickets=0

}

}

### **Using the Null Coalescing Operator (??)**

The program above can be simplified using the **null coalescing operator** (??). This operator returns the left-hand operand if it is not null; otherwise, it returns the right-hand operand.

#### **Example Program Using Null Coalescing Operator:**

using System;

class Program

{

static void Main()

{

int AvailableTickets;

int? TicketsOnSale = null;

// Using null coalescing operator ??

AvailableTickets = TicketsOnSale ?? 0; // If TicketsOnSale is null, assign 0

Console.WriteLine("Available Tickets={0}", AvailableTickets);

// Expected Output: Available Tickets=0

}

}

### **Explanation**

* **Without Null Coalescing Operator**: You need to write multiple lines of code to check if TicketsOnSale is null and then assign a value accordingly.
* **With Null Coalescing Operator**: The code is more concise and easier to read. The operator checks if TicketsOnSale is null, and if it is, AvailableTickets is assigned 0. Otherwise, AvailableTickets is assigned the value of TicketsOnSale.

### **Summary**

* **Nullable Types**: Allow value types to hold a null value, which is useful when dealing with database values that can be null.
* **Null Coalescing Operator (??)**: Simplifies code that assigns a default value if a nullable type is null.

# 06\_TypeCasting

### **1. Implicit Conversions**

An implicit conversion is done automatically by the compiler in situations where:

* There is no loss of information if the conversion is performed.
* There is no possibility of exceptions being thrown during the conversion.

**Example**: Converting an int to a float is an implicit conversion because it does not result in data loss and there is no risk of an exception.

#### **Implicit Conversion Example**

using System;

class Program

{

public static void Main()

{

int i = 100;

// float is a bigger data type than int.

// No loss of data or exceptions, so implicit conversion is allowed

float f = i;

Console.WriteLine(f);

// Expected Output: 100

}

}

* **Explanation**: In the above example, an int (i) is implicitly converted to a float (f). Since float can hold all integer values and more, there is no risk of data loss or exceptions.

### **2. Explicit Conversions**

An explicit conversion, also known as type casting, is required when:

* There is a risk of losing information (e.g., converting a float to an int loses the fractional part).
* There is a possibility of exceptions (e.g., overflow exceptions).

**Example**: Converting a float to an int requires an explicit conversion because the fractional part of the float is lost, and there's a risk of overflow.

#### **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. Explicit conversion is required.

int i = (int)f;

// OR use the Convert class

// int i = Convert.ToInt32(f);

Console.WriteLine(i);

// Expected Output: 100

}

}

* **Explanation**: In this example, the float value 100.25F is explicitly converted to an int using a cast (int)f. The fractional part (0.25) is lost, and the result is 100. Alternatively, you could use Convert.ToInt32(f).

### **3. Difference Between Parse() and TryParse()**

When dealing with strings that represent numeric values, you can use Parse() or TryParse() to convert the string to a numeric type.

* **Parse()**:
  + Converts a string to a numeric type (e.g., int.Parse("123")).
  + Throws an exception if the string cannot be converted (e.g., int.Parse("abc") would throw a FormatException).
* **TryParse()**:
  + Attempts to convert a string to a numeric type.
  + Returns true if the conversion is successful; otherwise, returns false.
  + Does not throw exceptions, making it safer to use when you're unsure if the string can be converted.

#### **Parse Example**

using System;

class Program

{

public static void Main()

{

string numberString = "123";

int number = int.Parse(numberString);

Console.WriteLine(number);

// Expected Output: 123

// This will throw an exception

// int invalidNumber = int.Parse("abc");

}

}

* **Explanation**: The string "123" is successfully parsed into an integer. However, trying to parse "abc" as an integer would throw a FormatException.

#### 

#### **TryParse Example**

using System;

class Program

{

public static void Main()

{

string numberString = "123";

int result;

bool isConversionSuccessful = int.TryParse(numberString, out result);

if (isConversionSuccessful)

{

Console.WriteLine(result);

// Expected Output: 123

}

else

{

Console.WriteLine("Conversion failed.");

}

// Safe conversion attempt with TryParse

string invalidNumberString = "abc";

isConversionSuccessful = int.TryParse(invalidNumberString, out result);

if (isConversionSuccessful)

{

Console.WriteLine(result);

}

else

{

Console.WriteLine("Conversion failed.");

// Expected Output: Conversion failed.

}

}

}

* **Explanation**: TryParse() attempts to convert "123" to an integer and succeeds, printing 123. When trying to convert "abc", TryParse() returns false, avoiding an exception and allowing you to handle the failure gracefully.

### **Summary**

* **Implicit Conversions**: Automatically handled by the compiler when there's no risk of data loss or exceptions.
* **Explicit Conversions**: Require manual casting or the use of the Convert class when there's a risk of data loss or exceptions.
* **Parse() vs. TryParse()**: Use Parse() when you are sure the string is a valid number. Use TryParse() for safer conversions when you cannot guarantee the string's validity.

# 07\_Arrays

### **Understanding Arrays in C#**

Arrays in C# are a fundamental way to store multiple values of the same type in a single data structure. They are particularly useful when you need to manage a fixed-size collection of elements.

### **1. Declaring and Initializing Arrays**

**Declaration and Initialization:**

* **Separate Declaration and Initialization**: You can declare an array and then assign values to its elements.
* **Combined Declaration and Initialization**: You can declare and initialize an array in a single line.

#### **Example Code**

using System;

class Program

{

public static void Main()

{

// Initialize and assign values in different lines

int[] EvenNumbers = new int[3];

EvenNumbers[0] = 0;

EvenNumbers[1] = 2;

EvenNumbers[2] = 4;

// Initialize and assign values in the same line

int[] OddNumbers = { 1, 3, 5 };

Console.WriteLine("Printing EVEN Numbers");

// Retrieve and print even numbers from the array

for (int i = 0; i < EvenNumbers.Length; i++)

{

Console.WriteLine(EvenNumbers[i]);

}

Console.WriteLine("Printing ODD Numbers");

// Retrieve and print odd numbers from the array

for (int i = 0; i < OddNumbers.Length; i++)

{

Console.WriteLine(OddNumbers[i]);

}

}

}

#### **Explanation**

* **Initialization**:
  + int[] EvenNumbers = new int[3]; declares an array with 3 elements.
  + EvenNumbers[0] = 0; sets the first element, and similarly for the others.
  + int[] OddNumbers = { 1, 3, 5 }; declares and initializes an array in one line.
* **Accessing Elements**:
  + EvenNumbers[i] and OddNumbers[i] access the elements at index i in their respective arrays.
  + Arrays are zero-based, meaning the first element is at index 0.

### **2. Advantages and Disadvantages of Arrays**

**Advantages**:

* **Strongly Typed**: Arrays in C# are strongly typed, meaning that all elements must be of the same data type.
* **Indexed Access**: Elements can be accessed quickly via their index.

**Disadvantages**:

* **Fixed Size**: Once an array is created, its size cannot be changed. If you need a dynamic size, you might need to use collections like List<T>.
* **Manual Indexing**: Arrays require you to manage indices manually, which can be error-prone.

### 

### 

### **Additional Notes**

**Multi-Dimensional Arrays**: C# supports multi-dimensional arrays (e.g., 2D arrays). For  
int[,] matrix = new int[2, 3]; // 2 rows, 3 columns

matrix[0, 0] = 1;

matrix[0, 1] = 2;

matrix[0, 2] = 3;

matrix[1, 0] = 4;

matrix[1, 1] = 5;

matrix[1, 2] = 6;

**Jagged Arrays**: Arrays of arrays, which allow for different sizes of inner arrays:  
  
int[][] jaggedArray = new int[2][];

jaggedArray[0] = new int[3] { 1, 2, 3 };

jaggedArray[1] = new int[2] { 4, 5 };

# 08\_If Statement

### **Understanding the if Statement in C#**

The if statement is one of the most fundamental control flow statements in C#. It allows you to execute a block of code conditionally based on whether a specified condition is true.

### **Basic Syntax**

if (condition)

{

// Code to execute if the condition is true

}

* **condition**: A boolean expression that evaluates to true or false.
* **Code Block**: The block of code within the curly braces {} is executed if the condition evaluates to true.

### **Example**

using System;

class Program

{

public static void Main()

{

int number = 10;

if (number > 0)

{

Console.WriteLine("The number is positive.");

}

}

}

* **Explanation**: The condition number > 0 is true because 10 is greater than 0. Therefore, the message "The number is positive." is printed.

### 

### **if-else Statement**

The if-else statement provides an alternative block of code to execute if the condition is false.

if (condition)

{

// Code to execute if the condition is true

}

else

{

// Code to execute if the condition is false

}

#### **Example**

using System;

class Program

{

public static void Main()

{

int number = -5;

if (number > 0)

{

Console.WriteLine("The number is positive.");

}

else

{

Console.WriteLine("The number is not positive.");

}

}

}

* **Explanation**: Since number is -5, which is not greater than 0, the else block is executed, printing "The number is not positive.".

### 

### **if-else if-else Statement**

The if-else if-else statement allows you to check multiple conditions sequentially.

if (condition1)

{

// Code to execute if condition1 is true

}

else if (condition2)

{

// Code to execute if condition2 is true

}

else

{

// Code to execute if none of the above conditions are true

}

#### **Example**

using System;

class Program

{

public static void Main()

{

int number = 0;

if (number > 0)

{

Console.WriteLine("The number is positive.");

}

else if (number < 0)

{

Console.WriteLine("The number is negative.");

}

else

{

Console.WriteLine("The number is zero.");

}

}

}

* **Explanation**: In this case, number is 0. The if condition is false, the else if condition is also false, so the final else block executes, printing "The number is zero.".

### **Nested if Statements**

You can also nest if statements within other if statements for more complex conditions.

#### **Example**

using System;

class Program

{

public static void Main()

{

int number = 15;

if (number >= 0)

{

if (number % 2 == 0)

{

Console.WriteLine("The number is non-negative and even.");

}

else

{

Console.WriteLine("The number is non-negative and odd.");

}

}

else

{

Console.WriteLine("The number is negative.");

}

}

}

* **Explanation**: The outer if checks if the number is non-negative. If true, the inner if checks if the number is even or odd.

## 

## Logical AND (&&) vs. Bitwise AND (&)

#### **Logical AND (&&)**

* **Usage**: Used to combine two boolean expressions.
* **Short-Circuiting**: The && operator performs short-circuit evaluation. This means if the first operand is false, the second operand is not evaluated because the result will be false regardless.

**Example**:  
  
bool a = true;

bool b = false;

if (a && b)

{

Console.WriteLine("Both are true.");

}

else

{

Console.WriteLine("At least one is false.");

}

* + In this case, the message "At least one is false." will be printed, and b will not be evaluated further if a is false.

#### **Bitwise AND (&)**

* **Usage**: Used to perform bitwise operations on integer types and can also be used for logical operations without short-circuiting.
* **Short-Circuiting**: The & operator does not perform short-circuit evaluation. Both operands are always evaluated.

**Example**:  
bool a = true;

bool b = false;

if (a & b)

{

Console.WriteLine("Both are true.");

}

else

{

Console.WriteLine("At least one is false.");

}

* + Here, both a and b are evaluated. The result will be "At least one is false.", but b is still evaluated even though a is true.

## Logical OR (||) vs. Bitwise OR (|)

#### **Logical OR (||)**

* **Usage**: Used to combine two boolean expressions.
* **Short-Circuiting**: The || operator performs short-circuit evaluation. If the first operand is true, the second operand is not evaluated because the result will be true regardless.

**Example**:  
  
bool a = true;

bool b = false;

if (a || b)

{

Console.WriteLine("At least one is true.");

}

else

{

Console.WriteLine("Both are false.");

}

* + In this case, the message "At least one is true." will be printed, and b will not be evaluated if a is true.

#### **Bitwise OR (|)**

* **Usage**: Used to perform bitwise operations on integer types and can also be used for logical operations without short-circuiting.
* **Short-Circuiting**: The | operator does not perform short-circuit evaluation. Both operands are always evaluated.

**Example**:  
  
bool a = true;

bool b = false;

if (a | b)

{

Console.WriteLine("At least one is true.");

}

else

{

Console.WriteLine("Both are false.");

}

* + Here, both a and b are evaluated. The result will be "At least one is true.", and b is still evaluated even though a is true.

### **Summary**

* **&& (Logical AND)**: Short-circuits. Evaluates the second operand only if necessary.
* **& (Bitwise AND)**: Does not short-circuit. Evaluates both operands.
* **|| (Logical OR)**: Short-circuits. Evaluates the second operand only if necessary.
* **| (Bitwise OR)**: Does not short-circuit. Evaluates both operands.

### **Examples Demonstrating Short-Circuiting**

using System;

class Program

{

public static void Main()

{

bool ShortCircuiting()

{

Console.WriteLine("ShortCircuiting called.");

return false;

}

bool NoShortCircuiting()

{

Console.WriteLine("NoShortCircuiting called.");

return true;

}

// Logical AND (&&) - Short-circuiting

if (ShortCircuiting() && NoShortCircuiting())

{

Console.WriteLine("Both are true.");

}

// Bitwise AND (&) - No short-circuiting

if (ShortCircuiting() & NoShortCircuiting())

{

Console.WriteLine("Both are true.");

}

}

}

* **Output**:
  + For &&, you will see "ShortCircuiting called." and not "NoShortCircuiting called.".
  + For &, you will see both "ShortCircuiting called." and "NoShortCircuiting called.".

# 09\_Switch Statement

The switch statement in C# provides a way to handle multiple possible values of a variable or expression more efficiently than a series of if-else statements. It is often used when you need to execute different blocks of code based on the value of a variable.

### **Basic Syntax**

switch (expression)

{

case value1:

// Code to execute if expression == value1

break;

case value2:

// Code to execute if expression == value2

break;

// Additional cases...

default:

// Code to execute if none of the cases match

break;

}

* **expression**: The variable or expression to be evaluated.
* **case value**: Each case specifies a value to compare against the expression. If it matches, the corresponding block of code is executed.
* **break**: Ends the current case block and exits the switch statement. Without break, execution will "fall through" to the next case.
* **default**: An optional block that executes if none of the case values match the expression.

### 

### 

### **Example**

Here’s a simple example that uses a switch statement to print the name of the day based on a number:

using System;

class Program

{

public static void Main()

{

int dayNumber = 3; // Let's say 1 = Monday, 2 = Tuesday, etc.

switch (dayNumber)

{

case 1:

Console.WriteLine("Monday");

break;

case 2:

Console.WriteLine("Tuesday");

break;

case 3:

Console.WriteLine("Wednesday");

break;

case 4:

Console.WriteLine("Thursday");

break;

case 5:

Console.WriteLine("Friday");

break;

case 6:

Console.WriteLine("Saturday");

break;

case 7:

Console.WriteLine("Sunday");

break;

default:

Console.WriteLine("Invalid day number");

break;

}

}

}

* **Explanation**: Given dayNumber is 3, the output will be "Wednesday". If dayNumber does not match any of the cases, the default block will execute, printing "Invalid day number".

### **Important Points**

1. **Expression Types**: The expression in the switch statement must evaluate to a value that can be compared with the case values. Typically, this includes int, char, string, and enumerations.
2. **Case Labels**: case labels must be unique. If two case labels have the same value, a compile-time error will occur.
3. **Fall-Through**: If you do not use break at the end of a case, execution will continue into the next case (fall-through). This is sometimes used intentionally.
4. **default Case**: The default block is optional but recommended as a catch-all for unexpected values.

### **Example with Fall-Through**

using System;

class Program

{

public static void Main()

{

int number = 5;

switch (number)

{

case 1:

case 2:

case 3:

Console.WriteLine("Number is between 1 and 3");

break;

case 4:

Console.WriteLine("Number is 4");

break;

default:

Console.WriteLine("Number is not between 1 and 4");

break;

}

}

}

* **Explanation**: The case 1, case 2, and case 3 blocks share the same code, demonstrating how fall-through can be used to handle multiple values with the same action.

### **Switch Expression (C# 8.0+)**

Starting with C# 8.0, you can use a switch expression which returns a value. It's more concise and allows for pattern matching.

using System;

class Program

{

public static void Main()

{

int dayNumber = 3;

string dayName = dayNumber switch

{

1 => "Monday",

2 => "Tuesday",

3 => "Wednesday",

4 => "Thursday",

5 => "Friday",

6 => "Saturday",

7 => "Sunday",

\_ => "Invalid day number"

};

Console.WriteLine(dayName);

}

}

* **Explanation**: The switch expression assigns the result to dayName. The \_ (discard) pattern is used to handle the default case.

# 10\_Loops

In C#, while and do-while loops are used for repeated execution of a block of code based on a condition. The key difference between them is in when the condition is checked:

## while Loop

The while loop evaluates the condition before the execution of the loop’s body. If the condition is true, the loop’s body executes. If the condition is false, the loop is not executed at all.

#### **Syntax**

while (condition)

{

// Code to execute while the condition is true

}

#### **Example**

using System;

class Program

{

public static void Main()

{

int count = 1;

// The loop will run as long as count is less than or equal to 5

while (count <= 5)

{

Console.WriteLine(count);

count++;

}

}

}

* **Explanation**: The while loop starts by checking if count is less than or equal to 5. As long as this condition is true, it prints the value of count and increments it. When count becomes 6, the condition becomes false and the loop stops.

## do-while Loop

The do-while loop evaluates the condition after the execution of the loop’s body. This ensures that the loop’s body is executed at least once, even if the condition is false on the first check.

#### **Syntax**

do

{

// Code to execute

} while (condition);

#### **Example**

using System;

class Program

{

public static void Main()

{

int count = 1;

// The loop will always run at least once

do

{

Console.WriteLine(count);

count++;

} while (count <= 5);

}

}

* **Explanation**: The do-while loop starts by executing the loop body first, and then checks if count is less than or equal to 5. As long as this condition is true, it continues to execute. If the condition is false initially, the loop still executes once.

### **Key Differences**

1. **Condition Check**:
   * **while Loop**: Condition is checked before executing the loop body. The loop body may not execute at all if the condition is false initially.
   * **do-while Loop**: Condition is checked after executing the loop body. The loop body always executes at least once.
2. **Use Case**:
   * **while Loop**: Use when the number of iterations is not known beforehand and the loop might not need to execute at all if the condition is not met initially.
   * **do-while Loop**: Use when you need the loop to execute at least once regardless of the condition.

### **Example with do-while Showing at Least One Execution**

using System;

class Program

{

public static void Main()

{

int number = 10;

// The loop will execute at least once

do

{

Console.WriteLine("This will print at least once.");

} while (number < 5); // The condition is false initially, but the body will still execute once

}

}

* **Explanation**: Even though number is 10 (and the condition number < 5 is false), the message "This will print at least once." will be printed once due to the do-while loop.

In C#, for and foreach loops are used for iterating over a collection of items or executing a block of code a specific number of times. Each has its own use cases and advantages.

## for Loop

The for loop is ideal when you know the number of iterations in advance or need to perform operations using an index.

#### **Syntax**

for (initialization; condition; iteration)

{

// Code to execute on each iteration

}

* **Initialization**: Executed once at the start of the loop.
* **Condition**: Evaluated before each iteration. If true, the loop body executes. If false, the loop terminates.
* **Iteration**: Executed after each iteration of the loop body.

#### **Example**

using System;

class Program

{

public static void Main()

{

// Print numbers from 1 to 5

for (int i = 1; i <= 5; i++)

{

Console.WriteLine(i);

}

}

}

* **Explanation**: The for loop starts with i initialized to 1, and continues as long as i is less than or equal to 5. After each iteration, i is incremented by 1.

#### 

#### **Advanced Example: Nested for Loops**

using System;

class Program

{

public static void Main()

{

// Print a multiplication table

for (int i = 1; i <= 3; i++)

{

for (int j = 1; j <= 3; j++)

{

Console.Write($"{i \* j} ");

}

Console.WriteLine();

}

}

}

* **Explanation**: This nested for loop prints a simple multiplication table. The outer loop iterates over rows, and the inner loop iterates over columns, calculating the product of i and j.

## foreach Loop

The foreach loop is used to iterate over collections or arrays. It’s particularly useful when you don’t need to modify the collection and just want to access each item.

#### **Syntax**

foreach (type variable in collection)

{

// Code to execute for each item in the collection

}

* **type**: The type of elements in the collection.
* **variable**: The variable that holds each item of the collection during iteration.
* **collection**: The collection or array you are iterating over.

#### **Example**

using System;

class Program

{

public static void Main()

{

int[] numbers = { 1, 2, 3, 4, 5 };

// Print each number in the array

foreach (int number in numbers)

{

Console.WriteLine(number);

}

}

}

* **Explanation**: The foreach loop iterates over each element in the numbers array, assigning each element to the number variable, and prints it.

#### **Advanced Example: Iterating Over a List**

using System;

using System.Collections.Generic;

class Program

{

public static void Main()

{

List<string> fruits = new List<string> { "Apple", "Banana", "Cherry" };

// Print each fruit in the list

foreach (string fruit in fruits)

{

Console.WriteLine(fruit);

}

}

}

* **Explanation**: The foreach loop iterates over each string in the fruits list and prints it.

### **Key Differences**

1. **Use Case**:
   * **for Loop**: Best used when you need to perform operations based on index or when the number of iterations is known in advance.
   * **foreach Loop**: Ideal for iterating over collections or arrays when you don’t need to modify the collection and are interested in processing each element.
2. **Index Access**:
   * **for Loop**: Provides direct access to the index of the current iteration, which allows you to perform operations based on the index.
   * **foreach Loop**: Does not provide index access and is read-only for the collection elements.
3. **Modification**:
   * **for Loop**: Can modify the collection being iterated over, though it's generally discouraged due to potential issues.
   * **foreach Loop**: Does not allow modification of the collection (elements can be modified if the collection allows it, but the collection itself cannot be modified).

# 

# 11\_Methods

### **Structure of a Method**

In C#, a method is a block of code that performs a specific task. It consists of several key components:

#### **Basic Structure of a Method**

[attributes] access-modifiers return-type method-name(parameters)

{

// Method Body: The code that defines what the method does

}

* **Attributes**: Special annotations that provide metadata about the method. We’ll discuss attributes in more detail later.
* **Access Modifiers**: Define the visibility of the method (e.g., public, private, protected). We’ll also explore these in more depth later.
* **Return Type**: The type of data the method returns (e.g., int, string, void). If the method doesn’t return anything, use void.
* **Method Name**: A meaningful name that describes what the method does.
* **Parameters**: A list of variables passed to the method. Parameters are optional.

#### **Example**

public int AddNumbers(int a, int b)

{

int sum = a + b;

return sum;

}

* **Access Modifier**: public
* **Return Type**: int
* **Method Name**: AddNumbers
* **Parameters**: int a, int b
* **Method Body**: Adds two numbers and returns the sum.

### **Static vs Instance Methods**

## Static Methods

* **Definition**: Methods defined with the static keyword.
* **Invocation**: Called using the class name, not an instance of the class.
* **Characteristics**:
  + Belong to the class itself rather than any object.
  + Can only access static members (variables or methods) of the class.
  + Useful for utility functions that don’t depend on object state.

#### **Example of Static Method**

public class Calculator

{

public static int Add(int a, int b)

{

return a + b;

}

}

// Calling the static method

int result = Calculator.Add(3, 5);

* **Explanation**: Add is a static method, so it’s called using the class name Calculator.

## Instance Methods

* **Definition**: Methods that do not have the static keyword.
* **Invocation**: Called on an instance (object) of the class.
* **Characteristics**:
  + Belong to the object created from the class.
  + Can access both static and instance members of the class.
  + Useful when the method needs to interact with the instance's state (its fields or properties).

#### 

#### 

#### 

#### 

#### **Example of Instance Method**

public class Calculator

{

public int Subtract(int a, int b)

{

return a - b;

}

}

// Creating an instance of the class

Calculator calc = new Calculator();

// Calling the instance method

int result = calc.Subtract(10, 3);

* **Explanation**: Subtract is an instance method, so it’s called on an instance of the Calculator class.

### **Key Differences**

1. **Invocation**:
   * **Static Method**: ClassName.MethodName()
   * **Instance Method**: ObjectName.MethodName()
2. **State Dependency**:
   * **Static Method**: Cannot access instance members directly.
   * **Instance Method**: Can access both static and instance members.
3. **Memory**:
   * **Static Method**: Exists once per class.
   * **Instance Method**: Exists for each instance of the class.

### **Summary**

* **Static methods** are used when the method logic doesn’t depend on the state of an object.
* **Instance methods** are used when the method needs to work with data stored in an instance of the class.

# 12\_Paramter Types

### **Method Parameter Types in C#**

In C#, methods can accept different types of parameters. These types define how the data is passed to the method and how changes inside the method affect the original variables.

#### **1. Value Parameters**

* **Behavior**: When you pass a value parameter to a method, a copy of the original data is made. Any modifications to this copy inside the method do not affect the original variable.
* **Use Case**: Use value parameters when you don’t want the method to change the original data.

**Example**:

public void IncrementValue(int x)

{

x = x + 1;

}

int a = 5;

IncrementValue(a);

Console.WriteLine(a); // Output: 5 (Original value remains unchanged)

#### **2. Reference Parameters (ref)**

* **Behavior**: When a reference parameter is passed using the ref keyword, the method works with the original variable instead of a copy. Changes made to the parameter inside the method will affect the original variable.
* **Use Case**: Use reference parameters when you need the method to modify the original variable.

**Example**:

public void IncrementValue(ref int x)

{

x = x + 1;

}

int a = 5;

IncrementValue(ref a);

Console.WriteLine(a); // Output: 6 (Original value is modified)

#### **3. Out Parameters (out)**

* **Behavior**: Out parameters allow a method to return more than one value. These parameters must be assigned a value inside the method before the method returns.
* **Use Case**: Use out parameters when you want to return multiple values from a method.

**Example**:

public void GetValues(out int x, out int y)

{

x = 5;

y = 10;

}

int a, b;

GetValues(out a, out b);

Console.WriteLine(a); // Output: 5

Console.WriteLine(b); // Output: 10

#### **4. Parameter Arrays (params)**

* **Behavior**: The params keyword allows a method to accept a variable number of arguments. You can pass a comma-separated list, an array, or no arguments at all.
* **Use Case**: Use parameter arrays when the number of arguments passed to the method may vary.

**Example**:

public void PrintNumbers(params int[] numbers)

{

foreach (int number in numbers)

{

Console.WriteLine(number);

}

}

PrintNumbers(1, 2, 3, 4); // Output: 1 2 3 4

PrintNumbers(5, 6); // Output: 5 6

PrintNumbers(); // Output: (No output)

### **Pass by Value vs. Pass by Reference**

#### **Pass by Value:**

* **Behavior**: When you pass by value, the method receives a copy of the variable. Changes made inside the method do not affect the original variable.
* **Example**: As seen in **Value Parameters**.

#### **Pass by Reference:**

* **Behavior**: When you pass by reference (using ref or out), the method works with the original variable. Changes made inside the method affect the original variable.
* **Example**: As seen in **Reference Parameters** and **Out Parameters**.

### **Summary**

* **Value Parameters**: Pass a copy, original remains unchanged.
* **Reference Parameters (ref)**: Pass the original, changes are reflected.
* **Out Parameters (out)**: Used to return multiple values, original must be assigned.
* **Parameter Arrays (params)**: Allow a variable number of arguments.

# 13\_Namespaces

**Namespaces** are a way to organize code elements like classes, interfaces, structs, enums, and delegates into a logical hierarchy. They help avoid name conflicts in large projects by grouping related functionalities together.

### **Key Concepts:**

1. **Namespaces and Assemblies**:
   * Namespaces do not correspond directly to file, directory, or assembly names. You can have classes belonging to the same namespace in different files or assemblies.
2. **Nesting Namespaces**:
   * Namespaces can be nested, meaning you can have a namespace inside another namespace. This allows for further organization and specificity.

**Example**:  
namespace ProjectA

{

namespace TeamA

{

class ClassA

{

public static void Print()

{

Console.WriteLine("This is Team A Print method");

}

}

}

namespace TeamB

{

class ClassA

{

public static void Print()

{

Console.WriteLine("This is Team B Print method");

}

}

}

1. **Avoiding Ambiguity**:
   * **Fully Qualified Names**: To avoid ambiguity when classes from different namespaces have the same name, you can use fully qualified names.

**Example**:  
using System;

class Demo

{

public static void Main()

{

// Avoid ambiguity by using the fully qualified name

ProjectA.TeamA.ClassA.Print();

ProjectA.TeamB.ClassA.Print();

}

}

* + **Alias Directives**: Another way to avoid ambiguity and improve readability is to use alias directives. You can assign a shorter name to a long namespace.

**Example**:  
  
using System;

using PTA = ProjectA.TeamA;

using PTB = ProjectA.TeamB;

class Demo

{

public static void Main()

{

// Avoid ambiguity by using alias directives

PTA.ClassA.Print();

PTB.ClassA.Print();

}

}

### 

### 

### **Namespace Members:**

A namespace can contain various members, including:

1. **Another Namespace**:
   * You can nest namespaces within each other to create a hierarchical structure.
2. **Class**:
   * Classes are the main building blocks of any C# application and can be organized within namespaces.
3. **Interface**:
   * Interfaces define a contract that classes can implement, and they can also be organized within namespaces.
4. **Struct**:
   * Structs are value types that can contain data and methods, and they can be placed inside namespaces.
5. **Enum**:
   * Enums are types that consist of a set of named constants, and they can also be organized within namespaces.
6. **Delegate**:
   * Delegates are types that represent references to methods with a specific signature, and they can be declared inside namespaces.

### **Summary:**

* **Namespaces**: Organize code and prevent name conflicts.
* **Fully Qualified Names**: Use when you need to distinguish between similar names in different namespaces.
* **Alias Directives**: Use to shorten long namespace names for readability.
* **Namespace Members**: Can include other namespaces, classes, interfaces, structs, enums, and delegates.

# 

# 

# 14\_Classes

### **Understanding Classes in C#**

**Classes** are fundamental building blocks in object-oriented programming (OOP) and are used to create complex custom data types in C#. Unlike simple data types like int, float, and double, classes allow you to model real-world entities by combining both data and behavior into a single unit.

### **Key Concepts of a Class:**

1. **Fields (Data)**:
   * Fields are variables that hold the data or state of the object. Each instance of a class can have its own set of fields with different values.

**Example**:  
class Car

{

public string make;

public string model;

public int year;

}

1. **Methods (Behavior)**:
   * Methods define the actions or behavior that an object of the class can perform. They operate on the data stored in the fields.

**Example**:  
class Car

{

public string make;

public string model;

public int year;

public void StartEngine()

{

Console.WriteLine("The engine is starting...");

}

}

## Constructors:

* **Constructors** are special methods used to initialize the fields of a class when an object is created. They have the same name as the class and do not have a return type.
* **Automatic Invocation**: A constructor is automatically called when you create a new instance of a class.
* **Default Constructor**: If you do not define a constructor, C# provides a default parameterless constructor automatically.

**Constructor Overloading**: You can have multiple constructors in a class with different numbers or types of parameters. This is called constructor overloading.  
**Example**:  
class Car

{

public string make;

public string model;

public int year;

// Constructor with parameters

public Car(string make, string model, int year)

{

this.make = make;

this.model = model;

this.year = year;

}

}

**Usage**:  
Car myCar = new Car("Toyota", "Corolla", 2020);

## Destructors:

* **Destructors** are special methods used to clean up resources before an object is removed from memory. In C#, destructors are less commonly used because the garbage collector handles memory management automatically.
* **Naming**: Destructors have the same name as the class but are prefixed with a ~ symbol.
* **No Parameters or Return Type**: Destructors cannot take parameters, and they do not return any value.

**Garbage Collection**: Destructors are usually called when the garbage collector determines that an object is no longer needed and can be removed from memory.

**Example**:

class Car

{

public string make;

public string model;

public int year;

// Destructor

~Car()

{

// Clean up code here

Console.WriteLine("Destructor called, cleaning up resources...");

}

}

### **Summary:**

* **Class**: A blueprint for creating objects that contain both data (fields) and behavior (methods).
* **Constructor**: A special method used to initialize an object's fields. It is automatically called when an object is created and can be overloaded.
* **Destructor**: A special method used to clean up resources. It is automatically called when the garbage collector decides to remove the object from memory.

Classes are essential for modeling real-world entities and organizing code in a way that promotes reusability and maintainability.

# 

# 15\_Static / Instance Members

### **Static and Instance Class Members in C#**

In C#, class members (fields, methods, properties, etc.) can be either **static** or **instance** members. Understanding the difference between these two types is crucial for managing the state and behavior of your classes effectively.

#### **Static Members:**

* **Definition**: A member with the static modifier is called a static member. These members belong to the class itself rather than to any specific instance of the class.
* **Invocation**: Static members are accessed using the class name, not through an object instance.
* **Memory**: Only one copy of a static member exists, regardless of how many instances of the class are created.

**Examples**: Static methods, static fields, static properties, static constructors, etc.  
**Example**:  
class MathUtility

{

public static double Pi = 3.14159;

public static double CalculateArea(double radius)

{

return Pi \* radius \* radius;

}

}

// Accessing static members using class name

double area = MathUtility.CalculateArea(5);

Console.WriteLine(MathUtility.Pi); // Output: 3.14159

#### **Instance Members:**

* **Definition**: A member without the static modifier is called an instance member. These members belong to individual instances (objects) of the class.
* **Invocation**: Instance members are accessed using an object of the class.
* **Memory**: Each instance of the class has its own copy of instance members, so each object can maintain its own state independently of other objects.

**Examples**: Instance methods, instance fields, instance properties, instance constructors, etc.  
**Example**:  
class Car

{

public string Make;

public string Model;

public void DisplayInfo()

{

Console.WriteLine($"Car Make: {Make}, Model: {Model}");

}

}

// Creating instances and accessing instance members

Car car1 = new Car { Make = "Toyota", Model = "Corolla" };

Car car2 = new Car { Make = "Honda", Model = "Civic" };

car1.DisplayInfo(); // Output: Car Make: Toyota, Model: Corolla

car2.DisplayInfo(); // Output: Car Make: Honda, Model: Civic

### **Key Differences:**

1. **Belonging**:
   * **Static Members**: Belong to the class itself.
   * **Instance Members**: Belong to specific instances of the class.
2. **Memory**:
   * **Static Members**: Only one copy exists, regardless of the number of instances.
   * **Instance Members**: Each instance of the class has its own copy.
3. **Invocation**:
   * **Static Members**: Accessed using the class name (ClassName.MemberName).
   * **Instance Members**: Accessed using an object instance (objectInstance.MemberName).
4. **Constructor**:
   * **Static Members**: Initialized via a static constructor, which is called only once for the entire class.
   * **Instance Members**: Initialized via instance constructors, which are called every time a new object is created.

### **Static Constructor:**

* **Purpose**: A static constructor is used to initialize static fields of a class. It is called automatically before any static member is accessed or any instance of the class is created.
* **Declaration**: Defined with the static keyword and does not take any parameters.
* **Call Frequency**: A static constructor is called only once, regardless of how many instances of the class are created.

**Execution Order**: The static constructor runs before any instance constructor or any static method is invoked.  
**Example**:  
  
class Logger

{

public static string LogFilePath;

// Static constructor

static Logger()

{

LogFilePath = "log.txt";

Console.WriteLine("Static constructor called");

}

public static void Log(string message)

{

// Logic to write message to log file

}

}

// Static constructor is called automatically before first use

Logger.Log("Application started");

### **Summary:**

* **Static Members**: Belong to the class, one copy exists, accessed via class name, initialized by static constructors.
* **Instance Members**: Belong to individual objects, multiple copies exist (one per object), accessed via object instances, initialized by instance constructors.
* **Static Constructors**: Used to initialize static members, called once before any static member is accessed or instance is created.

# 16\_Inheritance

### **Why Inheritance?**

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to create a new class (derived class) based on an existing class (base class). The derived class inherits fields, properties, and methods from the base class, allowing for code reuse and reducing redundancy. This approach helps in maintaining the code by grouping common functionalities in a single place.

### **Example Without Inheritance**

Consider two classes, FullTimeEmployee and PartTimeEmployee, both having common fields like FirstName, LastName, and Email, and methods like PrintFullName(). Without inheritance, these fields and methods would be duplicated in both classes.

public class FullTimeEmployee

{

public string FirstName;

public string LastName;

public string Email;

public float YearlySalary;

public void PrintFullName()

{

Console.WriteLine($"{FirstName} {LastName}");

}

}

public class PartTimeEmployee

{

public string FirstName;

public string LastName;

public string Email;

public float HourlyRate;

public void PrintFullName()

{

Console.WriteLine($"{FirstName} {LastName}");

### **Example With Inheritance**

By using inheritance, you can move the common code into a base class called Employee and then derive FullTimeEmployee and PartTimeEmployee from it.

public class Employee

{

public string FirstName;

public string LastName;

public string Email;

public void PrintFullName()

{

Console.WriteLine($"{FirstName} {LastName}");

}

}

public class FullTimeEmployee : Employee

{

public float YearlySalary;

}

public class PartTimeEmployee : Employee

{

public float HourlyRate;

}

### **Benefits of Inheritance**

1. **Code Reuse**: Common code is written once in the base class and reused in derived classes, which reduces code duplication.
2. **Maintainability**: Changes in common functionalities need to be made only in the base class, reducing the chances of errors.
3. **Specialization**: Derived classes can add or modify features specific to their requirements while inheriting the base class's common functionality.

### 

### **Realization and Specialization in Inheritance**

**Realization**: This refers to the implementation of methods defined in an interface or an abstract class by a derived class. The derived class "realizes" the interface or abstract class by providing concrete implementations of its members.  
  
public interface IEmployee

{

void Work();

}

public class FullTimeEmployee : IEmployee

{

public void Work()

{

Console.WriteLine("Working full-time.");

}

}

**Specialization**: This refers to the process of adding or overriding functionality in a derived class that extends the base class. The derived class specializes in certain behaviors that the base class does not provide.  
  
public class Employee

{

public void Work()

{

Console.WriteLine("Working...");

}

}

public class FullTimeEmployee : Employee

{

public new void Work()

{

Console.WriteLine("Working full-time with additional responsibilities.");

### **Inheritance Syntax**

public class ParentClass

{

// Parent Class Implementation

}

public class DerivedClass : ParentClass

{

// Child Class Implementation

}

* **Single Inheritance**: C# only supports single class inheritance, meaning a derived class can inherit from only one base class.
* **Multiple Interface Inheritance**: However, C# allows a class to inherit multiple interfaces.

### **Conclusion**

Inheritance simplifies code by promoting reuse and reducing redundancy, making your programs more modular and easier to maintain. Understanding and effectively utilizing inheritance is key to mastering object-oriented programming.

# 17\_Method Hiding

### **Inheritance Concepts: Hiding Base Class Members**

In C#, when a derived class declares a member with the same name as a member in its base class, the derived class member hides the base class member. This can lead to confusion if not handled properly, so C# provides the new keyword to explicitly indicate that you intend to hide the base class member.

If you do not use the new keyword when hiding a base class member, the compiler will issue a warning, but the code will still compile. This warning is a way of informing you that you might be unintentionally hiding a member from the base class.

### **Example of Hiding a Base Class Member**

public class BaseClass

{

public void Display()

{

Console.WriteLine("Display from BaseClass");

}

}

public class DerivedClass : BaseClass

{

public new void Display()

{

Console.WriteLine("Display from DerivedClass");

}

}

In this example, the Display method in DerivedClass hides the Display method in BaseClass. The new keyword is used to indicate that the hiding is intentional.

### 

### 

### **Ways to Invoke a Hidden Base Class Member**

**1. Using the base Keyword**The base keyword is used to refer to the base class within a derived class. You can use base to invoke a hidden member from the base class.  
  
public class DerivedClass : BaseClass

{

public new void Display()

{

base.Display(); // Calls the Display method from BaseClass

Console.WriteLine("Display from DerivedClass");

}

}

In this case, calling base.Display() will invoke the Display method from the BaseClass.

**2. Casting the Child Type to the Parent Type**You can cast an instance of the derived class to the base class type and then invoke the hidden member.

DerivedClass obj = new DerivedClass();

((BaseClass)obj).Display(); // Calls the Display method from BaseClass

This cast allows you to call the Display method from BaseClass even though DerivedClass has hidden it.

**3. Using a Parent Class Reference to a Child Class Object**You can create a reference of the base class type and assign it an object of the derived class. Then, use this reference to invoke the hidden member.  
  
BaseClass obj = new DerivedClass();

obj.Display(); // Calls the Display method from BaseClass

1. Since the reference is of type BaseClass, the Display method from BaseClass will be invoked.

### 

### **Summary**

* **new Keyword**: Explicitly indicates that a derived class member is hiding a member in the base class.
* **Invoking Hidden Members**:
  1. Use the base keyword to access the hidden member in the base class.
  2. Cast the derived class object to the base class type and invoke the hidden member.
  3. Use a base class reference to an object of the derived class to invoke the hidden member.

# 18\_Polymorphism

### **Polymorphism in C#**

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects to be treated as instances of their base class, enabling dynamic method invocation. This means that you can use a base class reference to call methods on derived class objects, and the correct method (i.e., the one in the derived class) will be executed at runtime.

Polymorphism promotes flexibility and reuse in code, as it allows for different implementations of a method to be defined in multiple derived classes, yet called through a common interface.

### **How Polymorphism Works**

To implement polymorphism in C#, you typically use the virtual and override keywords:

1. **virtual Keyword**:
   * In the base class, you declare a method as virtual to indicate that it can be overridden in any derived class.
   * The virtual keyword allows the derived class to provide its own specific implementation of the method.
2. **override Keyword**:
   * In the derived class, you use the override keyword to override the base class method with a new implementation.
   * This ensures that the derived class method will be invoked, even when called through a base class reference.

### **Example of Polymorphism**

Consider a simple example where we have a base class Animal and derived classes Dog and Cat.

using System;

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");

}

}

public class Program

{

public static void Main()

{

Animal myDog = new Dog();

Animal myCat = new Cat();

// Even though we are using the base class reference,

// the derived class method is called at runtime

myDog.MakeSound(); // Output: Dog barks

myCat.MakeSound(); // Output: Cat meows

}

}

### 

### 

### **Explanation**

* **Base Class (Animal)**:
  + The MakeSound method is declared as virtual, allowing it to be overridden in any derived class.
* **Derived Classes (Dog and Cat)**:
  + Both Dog and Cat override the MakeSound method using the override keyword to provide specific implementations.
* **Polymorphic Behavior**:
  + Even though myDog and myCat are declared as Animal types, at runtime, the MakeSound method of the actual object (Dog or Cat) is invoked. This is polymorphism in action.

### **Benefits of Polymorphism**

1. **Code Reusability**: Base classes can define general behavior, which can be overridden by derived classes to provide specific functionality.
2. **Flexibility**: You can write more flexible and extensible code. New derived classes can be added with minimal changes to existing code.
3. **Decoupling**: Polymorphism helps in decoupling the implementation from the code that uses it, making the codebase more maintainable.

### **Summary**

* **Polymorphism**: Allows you to use a base class reference to invoke derived class methods at runtime.
* **Virtual Methods**: Marked with virtual in the base class, indicating they can be overridden.
* **Overriding Methods**: Achieved using the override keyword in derived classes to provide specific implementations.

Polymorphism is a powerful OOP feature that enables dynamic behavior in your applications, allowing for more flexible and maintainable code.

# 19\_Overriding vs Hiding

### **Method Overriding vs Method Hiding in C#**

In C#, both **method overriding** and **method hiding** allow derived classes to provide a new implementation for a method defined in a base class. However, the key difference lies in how the method is invoked depending on the type of reference used.

### **1. Method Overriding**

* **Purpose**: Method overriding allows a derived class to provide a specific implementation of a method that is already defined in its base class.
* **Keywords**: The virtual keyword is used in the base class method, and the override keyword is used in the derived class method.
* **Behavior**: When a method is overridden, the derived class method is invoked even when using a base class reference pointing to a derived class object.

#### **Example of Method Overriding**

using System;

public class BaseClass

{

// Base class method marked as virtual

public virtual void Print()

{

Console.WriteLine("Base Class Print Method");

}

}

public class DerivedClass : BaseClass

{

// Derived class overrides the base class method

public override void Print()

{

Console.WriteLine("Derived Class Print Method");

}

}

public class Program

{

public static void Main()

{

BaseClass b = new DerivedClass();

b.Print(); // Output: "Derived Class Print Method"

}

}

### **Explanation:**

* The Print method in the BaseClass is marked as virtual, meaning it can be overridden.
* The DerivedClass overrides the Print method using the override keyword.
* When the Print method is called on the BaseClass reference (b), which points to a DerivedClass object, the overridden method in DerivedClass is executed.

### **2. Method Hiding**

* **Purpose**: Method hiding allows a derived class to define a new method with the same name as a method in the base class, effectively hiding the base class method.
* **Keywords**: The new keyword is used in the derived class method to indicate that it is hiding a method from the base class.
* **Behavior**: When a method is hidden, the method in the base class is invoked when using a base class reference pointing to a derived class object.

#### **Example of Method Hiding**

using System;

public class BaseClass

{

// Base class method

public virtual void Print()

{

Console.WriteLine("Base Class Print Method");

}

}

public class DerivedClass : BaseClass

{

// Derived class hides the base class method

public new void Print()

{

Console.WriteLine("Derived Class Print Method");

}

}

public class Program

{

public static void Main()

{

BaseClass b = new DerivedClass();

b.Print(); // Output: "Base Class Print Method"

}

}

### **Explanation:**

* The Print method in the DerivedClass is declared with the new keyword, indicating that it is hiding the base class method.
* When the Print method is called on the BaseClass reference (b), which points to a DerivedClass object, the base class method is executed, not the derived class method.

### 

### 

### 

### 

### 

### **Key Differences Between Method Overriding and Method Hiding:**

| **Feature** | **Method Overriding** | **Method Hiding** |
| --- | --- | --- |
| **Keywords Used** | virtual (in base class), override (in derived class) | new (in derived class) |
| **Behavior** | Derived class method is invoked even when using a base class reference. | Base class method is invoked when using a base class reference. |
| **Use Case** | To provide a specific implementation of a method that can be invoked polymorphically. | To replace a base class method with a new method in the derived class. |
| **Access to Base Class Method** | Can be accessed using base keyword in the derived class method. | Can be accessed by casting the derived object to the base class or using the base keyword. |

### **Conclusion**

* **Method Overriding** is used when you want a derived class to provide a specific implementation of a method that can be invoked using a base class reference, allowing for polymorphism.
* **Method Hiding** is used when you want to replace a base class method with a new method in the derived class, but be aware that base class references will still invoke the original base class method unless explicitly cast.

# 20\_Method Overloading

### **Method Overloading**

Method overloading, also known as function overloading, is a feature in C# that allows a class to have multiple methods with the same name but different signatures. The signature of a method includes the method name, the number, type, and kind (value, reference, or output) of parameters. It does **not** include the return type or the params modifier.

#### **Key Points:**

1. **Signature:** The method signature is what differentiates overloaded methods. It includes:
   * **Method Name:** The name of the method.
   * **Number of Parameters:** Different methods can have different numbers of parameters.
   * **Type of Parameters:** The data type of the parameters (e.g., int, float, string, etc.).
   * **Kind of Parameters:** The kind refers to whether parameters are passed by value, reference (ref), or output (out).
2. **Return Type is Not Part of the Signature:** Overloading cannot be achieved by changing just the return type of a method. For example, having two methods with the same name and parameters but different return types will cause a compilation error.
3. **Params Modifier:** The params modifier allows a method to accept a variable number of arguments, but it cannot be used alone to overload a method.

#### 

#### 

#### 

#### 

#### 

#### **Example:**

using System;

class Calculator

{

// Overloaded method with two int parameters

public int Add(int a, int b)

{

return a + b;

}

// Overloaded method with three int parameters

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

{

return a + b + c;

}

// Overloaded method with two float parameters

public float Add(float a, float b)

{

return a + b;

}

// Overloaded method with ref parameter

public int Add(ref int a, int b)

{

a += b;

return a;

}

// Overloaded method with out parameter

public void Add(int a, int b, out int result)

{

result = a + b;

}

}

class Program

{

static void Main()

{

Calculator calc = new Calculator();

// Calling overloaded methods

Console.WriteLine(calc.Add(5, 10)); // Calls Add(int, int) // 15

Console.WriteLine(calc.Add(5, 10, 15)); // Calls Add(int, int, int) // 30

Console.WriteLine(calc.Add(5.5f, 10.5f)); // Calls Add(float, float) // 16

int x = 5;

Console.WriteLine(calc.Add(ref x, 10)); // Calls Add(ref int, int) // 15

Console.WriteLine(x); // 15

int result;

calc.Add(5, 10, out result); // Calls Add(int, int, out int)

Console.WriteLine(result); // 15

}

}

In this example:

* The Add method is overloaded with different signatures.
* The method signature allows the compiler to differentiate between the various Add methods, ensuring that the correct one is called based on the arguments passed.

# 21\_Properties

In object-oriented programming, encapsulation is a fundamental concept that involves hiding the internal state of an object and allowing controlled access to that state through public methods or properties. This ensures that the internal data of the object cannot be directly accessed or modified in an uncontrolled way, which helps maintain the integrity and correctness of the object's state.

#### **Problems with Public Fields:**

If class fields are made public, they can be directly accessed and modified from outside the class. This can lead to several issues:

1. **Lack of Validation:** There is no control over what values are assigned to the fields. For example, a negative ID or a null name might be assigned, which could lead to invalid object states.
2. **Lack of Control:** You cannot enforce any rules or business logic on how fields are set or retrieved. For instance, if a Name is not provided, you might want to return "No Name" instead of null.
3. **Read-Only Constraints:** Some fields might need to be read-only after initialization, but public fields do not enforce this.

#### **Example of Problems with Public Fields:**

public class Student

{

public int ID;

public string Name;

public int PassMark;

}

public class Program

{

public static void Main()

{

Student student = new Student();

student.ID = -101; // ID should not be negative

student.Name = null; // Name should not be null

student.PassMark = 100; // PassMark should be read-only

Console.WriteLine($"ID: {student.ID}, Name: {student.Name}, PassMark: {student.PassMark}");

}

}

In this example, ID can be negative, Name can be null, and PassMark can be modified, leading to potential issues.

### **Properties: A Better Solution**

Properties provide a way to control how fields are accessed and modified. They allow you to define logic that is executed when the field is read (getter) or written to (setter). This gives you the ability to enforce rules and constraints on the field values, which helps maintain the integrity of your object's state.

#### **Example Using Properties:**

public class Student

{

private int id;

private string name;

private readonly int passMark = 35;

// Property for ID with validation

public int ID

{

get { return id; }

set

{

if (value <= 0)

throw new Exception("ID should be a positive number.");

id = value;

}

}

// Property for Name with validation

public string Name

{

get

{

if (string.IsNullOrEmpty(name))

return "No Name";

return name;

}

set

{

if (string.IsNullOrEmpty(value))

throw new Exception("Name cannot be null or empty.");

name = value;

}

}

// Read-only property for PassMark

public int PassMark

{

get { return passMark; }

}

}

public class Program

{

public static void Main()

{

Student student = new Student();

student.ID = 101; // Valid ID

student.Name = "John Doe"; // Valid Name

// student.PassMark = 100; // Compile-time error, PassMark is read-only

Console.WriteLine($"ID: {student.ID}, Name: {student.Name}, PassMark: {student.PassMark}");

}

}

#### **Advantages of Properties:**

1. **Validation:** You can validate the values being assigned to fields. For example, you can ensure that ID is always positive and Name is never null.
2. **Controlled Access:** You can control how fields are accessed and modified. For example, you can return "No Name" if the Name is not set.
3. **Read-Only Fields:** You can enforce read-only constraints by providing only a getter for certain fields (e.g., PassMark).

#### **Conclusion:**

Using properties instead of public fields allows you to enforce encapsulation, a key principle of object-oriented programming. This makes your code more robust, maintainable, and less error-prone, as you have full control over how data is accessed and modified within your classes.

### 

### 

### 

### 

### 

### 

### 

### 

### 

# 22\_Property Types

Properties in C# are a way to encapsulate and protect the fields in a class while providing a simple syntax to access and modify these fields. Properties use get and set accessors to define how fields are accessed or modified.

#### **1. Read/Write Property**

A property that includes both get and set accessors is a read/write property. This means the value can be read and modified.

public class Student

{

private int \_id;

// Read/Write property

public int ID

{

get { return \_id; }

set { \_id = value; }

}

}

#### **2. Read-Only Property**

A property that only has a get accessor is a read-only property. The value can only be read and not modified.

public class Student

{

private string \_name;

// Read-Only property

public string Name

{

get { return \_name; }

}

public Student(string name)

{

\_name = name; // Name is set through the constructor

}

}

#### **3. Write-Only Property**

A property that only has a set accessor is a write-only property. The value can be modified but not read.

public class Student

{

private int \_age;

// Write-Only property

public int Age

{

set { \_age = value; }

}

}

### **Auto-Implemented Properties**

Auto-implemented properties simplify property declarations when no additional logic is needed in the property accessors. With auto-implemented properties, you don't need to write the private field manually; the compiler automatically generates a private field.

#### **Example of Auto-Implemented Properties**

public class Student

{

// Auto-implemented property

public int ID { get; set; }

// Auto-implemented read-only property

public string Name { get; }

// Constructor to set Name

public Student(string name)

{

Name = name;

}

}

### **Key Points:**

1. **Get and Set Accessors:** Used to create properties that control access to a field.
2. **Read/Write Property:** Has both get and set accessors.
3. **Read-Only Property:** Has only a get accessor.
4. **Write-Only Property:** Has only a set accessor.
5. **Auto-Implemented Properties:** Simplifies property declaration by allowing the compiler to handle the backing field.

### **Advantages of Properties Over Traditional Get/Set Methods:**

* **Simpler Syntax:** Properties allow accessing the field as if it were public while maintaining encapsulation, making the code cleaner and easier to read.
* **Encapsulation:** Properties can include validation logic within the get and set accessors, providing more control over how fields are accessed or modified.

### **Summary:**

Properties in C# are powerful tools for managing how class fields are accessed and modified. Whether you need full control with custom logic or just simple data storage, properties offer a flexible way to encapsulate your class's internal data.

# 23\_Structs vs Class

### **Structs in C#**

Structs in C# are similar to classes in many ways but have some important differences. Like classes, structs can have private fields, public properties, constructors, and methods. However, structs are value types, whereas classes are reference types.

#### **Key Features of Structs**

1. **Private Fields**: Structs can contain private fields, which are used to store data internally.
2. **Public Properties**: Structs can have public properties to encapsulate and provide access to their private fields.
3. **Constructors**: Structs can have constructors to initialize their fields, but unlike classes, structs cannot have a parameterless constructor (unless it’s auto-implemented by the compiler).
4. **Methods**: Structs can contain methods to perform operations on their data.

#### **Example: Struct vs. Class Implementation**

##### **Class Implementation**

public class Customer

{

// Private Fields

private int id;

private string name;

// Constructor

public Customer(int Id, string Name)

{

this.id = Id;

this.name = Name;

}

// Public Properties

public int ID

{

get { return this.id; }

set { this.id = value; }

}

public string Name

{

get { return this.name; }

set { this.name = value; }

}

// Method

public void PrintName()

{

Console.WriteLine("ID: {0}, Name: {1}", this.id, this.name);

}

}

##### **Struct Implementation**

public struct Customer

{

// Private Fields

private int id;

private string name;

// Constructor

public Customer(int Id, string Name)

{

this.id = Id;

this.name = Name;

}

// Public Properties

public int ID

{

get { return this.id; }

set { this.id = value; }

}

public string Name

{

get { return this.name; }

set { this.name = value; }

}

// Method

public void PrintName()

{

Console.WriteLine("ID: {0}, Name: {1}", this.id, this.name);

}

}

#### **Differences Between Structs and Classes**

1. **Value Type vs. Reference Type**:
   * **Structs** are value types, meaning they are stored on the stack. When you assign a struct to a new variable, a copy of the struct is made.
   * **Classes** are reference types, meaning they are stored on the heap. Assigning a class object to a new variable only copies the reference, not the actual object.
2. **Memory Allocation**:
   * **Structs** are typically more memory-efficient because they avoid the overhead associated with heap allocation.
   * **Classes** involve heap allocation, which can incur additional overhead, especially when managing large numbers of objects.
3. **Inheritance**:
   * **Structs** do not support inheritance (except from interfaces), meaning you cannot derive one struct from another.
   * **Classes** support full inheritance, allowing for more complex object hierarchies.
4. **Default Constructor**:
   * **Structs** cannot have an explicit parameterless constructor (the compiler provides one by default).
   * **Classes** can have explicit parameterless constructors.
5. **Boxing and Unboxing**:
   * **Structs** may incur performance penalties when boxed/unboxed (converted to/from an object type), as this involves copying the value.
   * **Classes** do not have this issue, as they are reference types.

#### **Object Initializer Syntax**

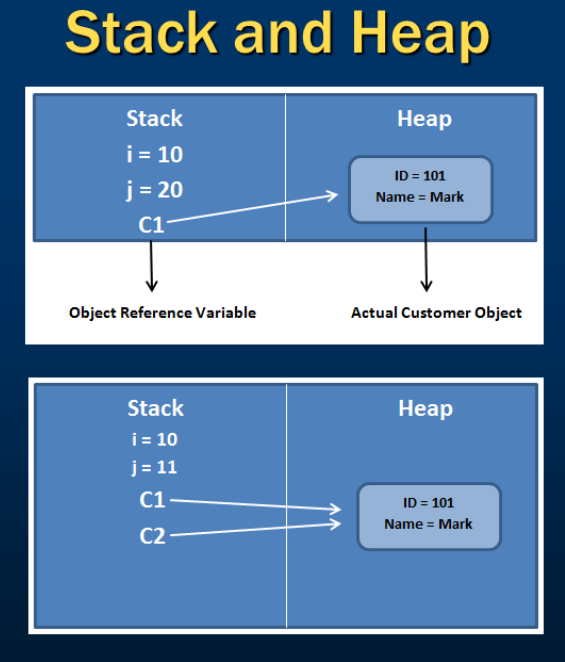
C# 3.0 introduced the object initializer syntax, which can be used with both structs and classes to initialize their fields/properties in a concise way.

Customer customer = new Customer { ID = 101, Name = "John Doe" };

This syntax can be used for both struct and class instances, providing a clean and readable way to initialize objects.

### **Conclusion**

Structs in C# are a lightweight alternative to classes, particularly useful when you need to create simple data structures that do not require the overhead of reference types or inheritance. Understanding when to use structs versus classes is essential for writing efficient and effective C# code.



### **Explanation:**

#### **Top Diagram:**

1. **Stack:**
   * The stack holds value types and references to objects.
   * Here, i and j are value types (like integers) and are directly stored on the stack with their values (i = 10 and j = 20).
   * C1 is a reference type, specifically an object reference variable. It holds the memory address (or reference) that points to an object located on the heap.
2. **Heap:**
   * The heap is where objects (instances of classes) are stored.
   * The actual object referred to by C1 is stored here. The object in this case has properties like ID = 101 and Name = Mark.
   * The reference in C1 on the stack points to this object on the heap.

#### **Bottom Diagram:**

1. **Stack:**
   * i remains the same with value 10, but j has been changed to 11.
   * C1 still holds the reference to the object on the heap.
   * A new reference variable C2 has been introduced. This is another object reference that also points to the same object on the heap as C1.
2. **Heap:**
   * The object on the heap is unchanged (ID = 101, Name = Mark).
   * Both C1 and C2 on the stack point to this same object on the heap.

### **Relation to Structs and Classes:**

* **Classes** are reference types, and thus objects of a class are stored on the heap. The stack only stores references (like C1 and C2) to these objects.
* **Structs** are value types, which means they are typically stored directly on the stack rather than the heap. This means that when you assign one struct to another, you copy the entire value rather than just a reference.

### **Key Points:**

* **Reference Types (Classes):** When you create an object of a class, the reference variable is stored on the stack, but the actual object is stored on the heap. Multiple reference variables (like C1 and C2) can point to the same object.
* **Value Types (Structs):** If this were dealing with structs, C1 and C2 would each hold a copy of the entire value (not just a reference), and any change to one would not affect the other. The object would be stored on the stack, not the heap.

### **Structs vs. Classes in C#**

In C#,Both structs and classes are used to create complex data types, but they have key differences that determine when each should be used. Let's explore these differences in detail.

#### **Similarities Between Structs and Classes**

* **Fields**: Both can have fields, which are variables that hold data.
* **Properties**: Both can have properties, which are used to control access to fields.
* **Constructors**: Both can have constructors, which are special methods used to initialize objects.
* **Methods**: Both can have methods, which define the behavior of the object.
* **Interfaces**: Both can implement interfaces.

#### **Key Differences Between Structs and Classes**

1. **Value Type vs. Reference Type**:
   * **Structs** are value types, meaning they store the actual data. When you assign one struct to another, a copy of the data is made.

public struct Point

{

public int X { get; set; }

public int Y { get; set; }

}

public class Program

{

public static void Main()

{

Point point1 = new Point { X = 10, Y = 20 };

Point point2 = point1; // A copy of the data is made

point2.X = 30;

Console.WriteLine(point1.X); // Outputs: 10 (unchanged)

Console.WriteLine(point2.X); // Outputs: 30 (different copy)

}

}

* + **Classes** are reference types, meaning they store a reference (or pointer) to the actual data. When you assign one class instance to another, both variables reference the same object.

public class Person

{

public string Name { get; set; }

}

public class Program

{

public static void Main()

{

Person person1 = new Person();

person1.Name = "Alice";

Person person2 = person1; // person2 references the same object as person1

person2.Name = "Bob";

Console.WriteLine(person1.Name); // Outputs: Bob

}

}

1. **Memory Allocation**:
   * **Structs** are typically stored on the stack, making them faster to access. However, this also means they are destroyed as soon as they go out of scope.
   * **Classes** are stored on the heap, which is managed by the garbage collector. They persist in memory until they are no longer referenced and are then cleaned up by the garbage collector.
2. **Scope and Lifetime**:
   * **Value types (Structs)**: The value type is destroyed when it goes out of scope, meaning it only exists for the duration of the method or block in which it is declared.
   * **Reference types (Classes)**: The reference variable is destroyed when it goes out of scope, but the object it points to on the heap persists until the garbage collector removes it.
3. **Copying Behavior**:
   * **Structs**: When a struct is copied, all the data is copied. Changes to the new struct do not affect the original struct.
   * **Classes**: When a class instance is copied, only the reference is copied, meaning both the original and the new reference point to the same object. Changes to the object via either reference affect the same underlying object.
4. **Inheritance**:
   * **Structs** cannot inherit from another class or struct. They are sealed types, meaning they cannot be inherited by another class or struct.
   * **Classes** can inherit from other classes, allowing for a hierarchy of types. Classes can be inherited unless they are explicitly marked as sealed.
5. **Destructors**:
   * **Structs** cannot have destructors. They are not needed because structs are value types and are automatically cleaned up when they go out of scope.
   * **Classes** can have destructors, which are used to clean up resources when the object is destroyed. This is especially useful for managing unmanaged resources like file handles or database connections.
6. **Parameterless Constructors**:
   * **Structs** cannot have explicit parameterless constructors. The compiler provides a default constructor that initializes all fields to their default values.
   * **Classes** can have explicit parameterless constructors to initialize fields with specific values.

#### **Examples of Structs in the .NET Framework**

* **int (System.Int32)**
* **double (System.Double)**
* **DateTime (System.DateTime)**

These are all examples of structs because they are simple types that benefit from the performance advantages of being value types.

#### **Note on Inheritance**

* A **class** or **struct** cannot inherit from another struct. Structs are sealed types by design to prevent inheritance.
* **Classes** can be marked as sealed to prevent inheritance. The sealed keyword ensures that no other class can derive from that class, which can be useful for design or performance reasons.

### **Summary**

Structs are best used for small, simple objects that represent a single value or a small group of related values. They are ideal for performance-critical applications where value semantics (copying the actual data) are important. Classes, on the other hand, are more suitable for complex objects that require reference semantics, inheritance, and more sophisticated memory management.

# 24\_Interfaces

### **Understanding Interfaces in C#**

Interfaces are a fundamental concept in object-oriented programming, especially in C#. They provide a way to define contracts that classes or structs can implement. Let's dive deeper into how interfaces work, including some examples.

### **1. Creating Interfaces**

* Interfaces in C# are defined using the interface keyword.
* An interface can include declarations for methods, properties, events, and indexers.
* However, unlike classes, interfaces do not provide any implementation for these members.

public interface IVehicle

{

void StartEngine();

void StopEngine();

int Speed { get; set; }

}

### **2. No Implementation in Interfaces**

* Interfaces only declare members; they do not provide implementation.
* Attempting to implement any member within the interface itself will result in a compile-time error.

// Incorrect: Implementation inside interface is not allowed

public interface IVehicle

{

void StartEngine()

{

// Compilation error

Console.WriteLine("Engine started.");

}

}

### 

### **3. Public Members by Default**

* All members of an interface are public by default.
* Explicit access modifiers (like public, private, protected) are not allowed in interfaces.

public interface IDatabase

{

void Connect(); // This is implicitly public

}

### **4. No Fields in Interfaces**

* Interfaces cannot contain fields (variables). They can only have methods, properties, events, and indexers.

public interface IExample

{

// int myField; // Not allowed in interfaces

void MyMethod();

}

### **5. Implementing Interfaces**

* When a class or struct implements an interface, it must provide implementations for all the interface members. Failure to do so will result in a compilation error.

public class Car : IVehicle

{

public int Speed { get; set; }

public void StartEngine()

{

Console.WriteLine("Car engine started.");

}

public void StopEngine()

{

Console.WriteLine("Car engine stopped.");

### **6. Multiple Interface Inheritance**

* A class or struct can implement multiple interfaces. This is a powerful feature as it allows a class to adhere to multiple contracts.
* However, a class cannot inherit from more than one class (no multiple inheritance for classes).

public interface IFlyable

{

void Fly();

}

public interface ISwimmable

{

void Swim();

}

public class AmphibiousVehicle : IFlyable, ISwimmable

{

public void Fly()

{

Console.WriteLine("Vehicle is flying.");

}

public void Swim()

{

Console.WriteLine("Vehicle is swimming.");

}

}

### **7. Interface Inheritance**

* Interfaces can inherit from other interfaces. A class implementing a derived interface must implement all members from the entire interface inheritance chain.

public interface IShape

{

void Draw();

}

public interface IColoredShape : IShape

{

string Color { get; set; }

}

public class Circle : IColoredShape

{

public string Color { get; set; }

public void Draw()

{

Console.WriteLine($"Drawing a {Color} circle.");

}

}

### **8. Interface Reference Variables**

* You cannot instantiate an interface directly, but an interface reference variable can point to an object of a class that implements the interface.

public interface IAnimal

{

void Speak();

}

public class Dog : IAnimal

{

public void Speak()

{

Console.WriteLine("Bark!");

}

}

public class Program

{

public static void Main()

{

IAnimal myDog = new Dog();

myDog.Speak(); // Outputs: Bark!

}

}

### **9. Explicit Interface Implementation**

* When a class implements multiple interfaces that have the same method name, explicit interface implementation allows the class to distinguish between the methods.
* With explicit implementation, the interface method can only be accessed through an interface reference, not a class instance.

public interface IPrinter

{

void Print();

}

public interface IScanner

{

void Print(); // Same method name as in IPrinter

}

public class MultiFunctionDevice : IPrinter, IScanner

{

void IPrinter.Print()

{

Console.WriteLine("Printing from Printer interface.");

}

void IScanner.Print()

{

Console.WriteLine("Printing from Scanner interface.");

}

}

public class Program

{

public static void Main()

{

MultiFunctionDevice device = new MultiFunctionDevice();

// Accessing the methods explicitly through the interface references

((IPrinter)device).Print(); // Outputs: Printing from Printer interface.

((IScanner)device).Print(); // Outputs: Printing from Scanner interface.

}

}

### **10. Default & Explicit Implementation**

* If a class implements multiple interfaces with the same method names, it can choose to provide a default implementation for one and explicit implementation for others.

public interface IFirstInterface

{

void Show();

}

public interface ISecondInterface

{

void Show();

}

public class MyClass : IFirstInterface, ISecondInterface

{

public void Show() // Default implementation for IFirstInterface

{

Console.WriteLine("Default implementation for IFirstInterface");

}

void ISecondInterface.Show() // Explicit implementation for ISecondInterface

{

Console.WriteLine("Explicit implementation for ISecondInterface");

}

}

public class Program

{

public static void Main()

{

MyClass obj = new MyClass();

obj.Show(); // Calls default implementation

((ISecondInterface)obj).Show(); // Calls explicit implementation

}

}

### **Summary:**

* **Interfaces** define contracts that classes or structs must adhere to.
* They enable **polymorphism** and **multiple inheritance** of behaviors in C#.
* **Explicit interface implementation** allows handling of naming conflicts when implementing multiple interfaces.

# 25\_Abstract Classes

### **Understanding Abstract Classes in C#**

Abstract classes in C# are a way to define a class that cannot be instantiated on its own and is meant to serve as a base class for other classes. Let's break down what abstract classes are, how they compare to interfaces, and provide examples for each point.

### **1. Creating Abstract Classes**

* The abstract keyword is used to create an abstract class.
* An abstract class is incomplete, which means you cannot create an instance of an abstract class directly.

public abstract class Shape

{

public abstract void Draw(); // Abstract method with no implementation

}

### **2. Abstract Classes Cannot Be Instantiated**

* Since abstract classes are incomplete, they cannot be instantiated.

// Shape shape = new Shape(); // This will result in a compile-time error

### **3. Abstract Classes as Base Classes**

* Abstract classes are primarily used as base classes. They provide a common definition of a base class that multiple derived classes can share.

public class Circle : Shape

{

public override void Draw()

{

Console.WriteLine("Drawing a Circle");

}

}

### **4. Abstract Classes Cannot Be Sealed**

* Abstract classes cannot be marked as sealed. A sealed class cannot be inherited, which conflicts with the purpose of abstract classes.

public abstract class Shape

{

// Implementation here...

}

// public sealed abstract class Shape { } // This will result in a compile-time error

### **5. Abstract Members in Abstract Classes**

* An abstract class may contain abstract members such as methods, properties, indexers, and events, but it is not mandatory.
* Abstract members are declared without any implementation and must be implemented in derived classes.

public abstract class Vehicle

{

public abstract void StartEngine(); // Abstract method

public void StopEngine() // Non-abstract method with implementation

{

Console.WriteLine("Engine stopped.");

}

}

### **6. Derived Classes and Abstract Members**

* If a class inherits an abstract class, it must provide implementations for all inherited abstract members.
* If the derived class does not implement all abstract members, it must be marked as abstract.

public class Car : Vehicle

{

public override void StartEngine()

{

Console.WriteLine("Car engine started.");

}

}

public abstract class ElectricVehicle : Vehicle

{

// This class remains abstract because it does not implement StartEngine

}

### **7. Abstract Classes vs Interfaces**

| **Abstract Classes** | **Interfaces** |
| --- | --- |
| Can have implementations for some members. | Cannot have implementations for any members. |
| Can have fields (variables). | Cannot have fields. |
| Can inherit from other abstract classes or interfaces. | Can inherit from other interfaces only. |
| A class can inherit from only one abstract class. | A class can inherit from multiple interfaces. |
| Abstract class members can have access modifiers. | Interface members are public by default and cannot have access modifiers. |

### **8. Examples**

#### **Example 1: Abstract Class with Abstract and Non-Abstract Members**

public abstract class Animal

{

public abstract void Speak(); // Abstract method

public void Eat() // Non-abstract method

{

Console.WriteLine("Eating...");

}

}

public class Dog : Animal

{

public override void Speak()

{

Console.WriteLine("Bark!");

}

}

class Program

{

static void Main()

{

Animal myDog = new Dog();

myDog.Speak(); // Outputs: Bark!

myDog.Eat(); // Outputs: Eating...

}

}

#### **Example 2: Interface vs Abstract Class**

public interface IMovable

{

void Move(); // Interface method (no implementation)

}

public abstract class Transport

{

public int Speed { get; set; } // Field with property

public abstract void Start(); // Abstract method

}

public class Car : Transport, IMovable

{

public override void Start()

{

Console.WriteLine("Car is starting.");

}

public void Move()

{

Console.WriteLine("Car is moving.");

}

}

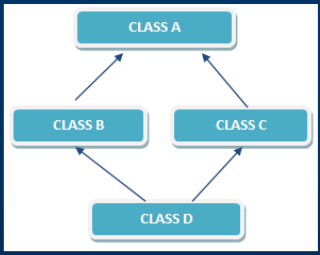
### **Summary:**

* **Abstract classes** provide a base for other classes to build upon, and they can contain both abstract (unimplemented) and non-abstract (implemented) members.
* **Interfaces** define a contract that classes must adhere to, but they do not provide any implementation.
* Use **abstract classes** when you need to share code between several closely related classes.
* Use **interfaces** when you want to define a contract that can be implemented by any class, regardless of its place in the class hierarchy.

# 26\_Multiple Inheritance

### **The Diamond Problem in Multiple Inheritance**

In object-oriented programming, the "Diamond Problem" is a well-known issue that arises when a class inherits from two classes that both inherit from a common base class. C# doesn't support multiple inheritance for classes to avoid this problem, but the concept can be understood through interfaces.



### **Diamond Problem Scenario:**

1. **Class Structure:**
   * **Class A**: A base class with a virtual method Print().
   * **Class B** and **Class C**: Both inherit from **Class A** and override the Print() method.
   * **Class D**: Hypothetically inherits from both **Class B** and **Class C**.
2. **The Problem:**
   * If **Class D** calls the Print() method without overriding it, there's ambiguity about whether it should inherit the method from **Class B** or **Class C**. This is the core of the diamond problem.

### **C# and Multiple Inheritance**

C# does not allow multiple inheritance for classes precisely to avoid such ambiguities. However, it does allow a class to implement multiple interfaces, which can simulate multiple inheritance in some scenarios.

### 

### **Example of Diamond Problem with Classes (Hypothetical in C#)**

Since C# doesn't support multiple class inheritance, here's a hypothetical example:

class A

{

public virtual void Print()

{

Console.WriteLine("Class A Implementation");

}

}

class B : A

{

public override void Print()

{

Console.WriteLine("Class B Overriding Print Method");

}

}

class C : A

{

public override void Print()

{

Console.WriteLine("Class C Overriding Print Method");

}

}

// Class D : B, C // C# does not allow this, so it's commented out

// {

// // In a language that allows this, there would be ambiguity here.

// }

### 

### 

### 

### **Multiple Interface Inheritance**

C# allows multiple inheritance with interfaces, which can create a similar scenario without ambiguity because interfaces do not have implementations.

### **Example: Multiple Interface Inheritance**

using System;

interface IA

{

void AMethod();

}

interface IB

{

void BMethod();

}

class A : IA

{

public void AMethod()

{

Console.WriteLine("A");

}

}

class B : IB

{

public void BMethod()

{

Console.WriteLine("B");

}

}

class AB : IA, IB

{

A a = new A();

B b = new B();

public void AMethod()

{

a.AMethod();

}

public void BMethod()

{

b.BMethod();

}

}

class Program

{

public static void Main()

{

AB ab = new AB();

ab.AMethod(); // Output: A

ab.BMethod(); // Output: B

}

}

### **Key Points:**

* **Diamond Problem:** Arises when a class inherits from two classes that each inherit from the same base class, leading to ambiguity in method resolution. This is called the Diamond Problem.
* **C# Solution:** C# avoids this problem by not supporting multiple inheritance for classes. Instead, you can use interfaces for a similar effect without method implementation ambiguity.
* **Interfaces:** Multiple interfaces can be implemented by a class, and each interface can be implemented separately or delegated to other class instances, as shown in the AB class example.

### **Conclusion**

The Diamond Problem is a significant reason why C# restricts multiple inheritance for classes. By using interfaces, C# allows for multiple inheritance in a controlled manner, avoiding the ambiguities that arise with method resolution in traditional multiple inheritance scenarios.

# 27\_Inner Classes

In C#The concept of inner classes is similar to Java, but with some differences in syntax and usage. Let's break down how you can implement the example you provided in C#.

using System;

class A

{

int marks;

public void Show()

{

Console.WriteLine("In show");

}

public class B

{

public void Config()

{

Console.WriteLine("In config of B");

}

}

public static class C

{

public void Config()

{

Console.WriteLine("In config of C");

}

}

}

class Inner

{

public static void Main(string[] args)

{

// Create an instance of the outer class

A obj = new A();

obj.Show();

// Create an instance of the inner class B

A.B obj1 = new A.B();

obj1.Config();

// Create an instance of the static inner class C

A.C obj2 = new A.C();

obj2.Config();

}

}

### **Explanation:**

1. **Class A**: The outer class in C#. It has a method Show() which is similar to Java's show().
2. **Class B (Non-static Inner Class)**:
   * In C#, inner classes do not need to be static to be instantiated. Unlike Java, you don't have to reference the outer class's instance to create an instance of the inner class.
   * In the example above, A.B obj1 = new A.B(); works directly, unlike Java where you have to create it using obj.new B().
3. **Class C (Static Inner Class)**:
   * Static inner classes in C# can be instantiated directly using A.C obj2 = new A.C();, which is very similar to Java's syntax for static inner classes.
   * Static inner classes cannot access the non-static members of the outer class directly.

### **Key Differences:**

* In Java, non-static inner classes require an instance of the outer class to be instantiated. In C#, you can instantiate an inner class directly without needing an instance of the outer class, but they work similarly to non-static inner classes in Java.
* Static inner classes in both languages are quite similar in terms of instantiation and usage.

This C# example mirrors your Java example, demonstrating how to use inner classes, both static and non-static, in C#.

# 28\_Annonymous Inner Classes

In C#,anonymous inner classes can be achieved using **anonymous methods** or **lambda expressions** for interfaces and abstract classes. Here's how you can achieve the functionality described in your Java example:

### **1. Overriding a method in an anonymous class (equivalent to Java's show method override):**

In C#, you would typically use a delegate or an anonymous method to achieve this. However, to simulate the behavior of an anonymous inner class directly within a class, you would use inheritance or interfaces with a similar approach.

### **2. Implementing an abstract class method using an anonymous class**

This is similar in C# by using an anonymous method or lambda expression.

### **Example in C#:**

using System;

class A

{

public virtual void Show()

{

Console.WriteLine("In Outer class show");

}

}

abstract class C

{

public abstract void Config();

}

class Program

{

static void Main(string[] args)

{

// Anonymous class to override the Show method in class A

A obj = new A()

{

public override void Show()

{

Console.WriteLine("In Inner class show");

}

};

obj.Show();

// Anonymous method to implement the abstract Config method in class C

C obj1 = new C()

{

public override void Config()

{

Console.WriteLine("In Inner class Config");

}

};

obj1.Config();

}

}

### **Explanation:**

* **Anonymous Class (Override Show Method):**
  + In the Main method, an instance of class A is created, and the Show method is overridden in an anonymous class.
  + In C#, this is achieved by overriding the method directly when creating the object.
* **Anonymous Method for Abstract Class:**
  + An instance of the abstract class C is created, and its abstract method Config is implemented using an anonymous method.

This approach simulates the behavior of Java's anonymous inner classes in C#. While C# does not have a direct equivalent to Java's inner classes, these methods achieve similar functionality.

# 29\_Delegates

In C#, you can achieve Functional Interfaces using **delegates** and **lambda expressions**.

### **Translating the Java Example to C#:**

Here's how you can achieve the same in C#:

1. **Functional Interface in Java**: This is equivalent to a delegate in C#.
2. **Lambda Expressions in Java**: These work similarly in C# and are often used with delegates or directly with LINQ.

### **Example in C#:**

using System;

public delegate void A(int i);

public delegate int B(int a, int b);

class FunctionInterf

{

static void Main(string[] args)

{

// Using delegate A with lambda expression

A obj = i => Console.WriteLine("In show " + i);

obj(5);

// Using delegate B with lambda expression

B obj1 = (a, b) => a + b;

Console.WriteLine(obj1(7, 8));

}

}

### **Explanation:**

1. **Delegate A**:
   * In C#, the A delegate is equivalent to your Java functional interface A.
   * The delegate A is defined to take an integer parameter and return void.
   * The lambda expression i => Console.WriteLine("In show " + i) provides an implementation for the delegate.
2. **Delegate B**:
   * The delegate B is equivalent to your Java functional interface B.
   * The delegate B is defined to take two integers as parameters and return an integer.
   * The lambda expression (a, b) => a + b provides an implementation that adds the two integers.
3. **Usage**:
   * You create instances of these delegates and provide implementations via lambda expressions.
   * The lambda expressions are passed as the implementation of these delegates.

### **Key Points:**

* **Delegates**: In C#, a delegate is a type that represents references to methods with a specific parameter list and return type.
* **Lambda Expressions**: In C#, lambda expressions are used to create anonymous methods. They can be used wherever a delegate type is expected.
* **SAM (Single Abstract Method)**: In Java, functional interfaces can have only one abstract method, which makes them compatible with lambda expressions. In C#, this is naturally handled by delegates.

### **Notes:**

* In the C# For example, the delegate keyword is used to define A and B. They work similarly to the functional interfaces in Java.
* Lambda expressions in C# are powerful and can be used to succinctly define the behavior you want to pass around as a method reference.

This approach allows you to mimic the behavior of Java's functional interfaces using delegates in C#.

# 30\_Exception Handling

In C#, exception handling is a crucial aspect of writing robust and reliable code. Exceptions are errors that occur during the execution of a program. These errors can range from attempting to access a file that doesn't exist to trying to divide a number by zero. Here's a breakdown of the key concepts you've mentioned, along with a C# example.

### **Key Concepts**

1. **Exception Class Hierarchy**:
   * In C#, exceptions are represented as classes that derive from the System.Exception base class. This base class provides essential properties such as:
     + Message: A description of the error.
     + StackTrace: A string representation of the immediate frames on the call stack.
2. **Handling Exceptions**:
   * **try**: The code that might throw an exception is placed inside the try block.
   * **catch**: The catch block is used to handle the exception. You can have multiple catch blocks for different exception types.
   * **finally**: The finally block is optional and is used to clean up resources, like closing file streams or database connections, whether or not an exception occurred.
3. **Best Practices**:
   * Always catch specific exceptions first before catching the base Exception type.
   * Use the finally block to release resources to ensure that they are freed even if an exception occurs.
4. **Inner Exceptions**:
   * An InnerException is the exception that caused the current exception. This is useful for debugging nested exceptions, where one exception leads to another.

### **C# Example**

using System;

using System.IO;

class ExceptionHandlingExample

{

static void Main()

{

try

{

// Simulate a file read operation that can cause an exception

ReadFile("nonexistentfile.txt");

}

catch (FileNotFoundException ex)

{

Console.WriteLine("File not found exception caught:");

Console.WriteLine($"Message: {ex.Message}");

Console.WriteLine($"Stack Trace: {ex.StackTrace}");

}

catch (Exception ex) // General catch block for any other exceptions

{

Console.WriteLine("An exception occurred:");

Console.WriteLine($"Message: {ex.Message}");

Console.WriteLine($"Stack Trace: {ex.StackTrace}");

// Check for inner exception

if (ex.InnerException != null)

{

Console.WriteLine("Inner Exception:");

Console.WriteLine($"Message: {ex.InnerException.Message}");

Console.WriteLine($"Type: {ex.InnerException.GetType()}");

}

}

finally

{

Console.WriteLine("Finally block executed.");

}

}

static void ReadFile(string filePath)

{

try

{

// Attempt to read a file that does not exist

string content = File.ReadAllText(filePath);

Console.WriteLine(content);

}

catch (Exception ex)

{

// Throwing a new exception with the current exception as the inner exception

throw new Exception("An error occurred while reading the file.", ex);

}

}

}

### **Explanation of the Code**

1. **Simulated Exception**:
   * The ReadFile method tries to read a file that doesn't exist, which throws a FileNotFoundException.
2. **Multiple Catch Blocks**:
   * The catch (FileNotFoundException ex) block catches the specific FileNotFoundException and outputs details about the error.
   * The catch (Exception ex) block is a general catch block that catches any other exceptions that might occur.
3. **Inner Exception**:
   * Inside the ReadFile method, a new exception is thrown with the original exception as an inner exception. This helps in preserving the original error context.
4. **Finally Block**:
   * The finally block executes regardless of whether an exception was thrown or not. It is used here to demonstrate that resource cleanup would occur here if needed.

### **Summary**

* Exception handling in C# allows you to manage errors gracefully and ensures your program can deal with unforeseen issues without crashing.
* The use of try, catch, and finally blocks enables you to catch specific errors, clean up resources, and handle nested exceptions using inner exceptions.
* Always remember to prioritize specific exceptions before the general Exception type to avoid missing more detailed exception handling opportunities.

# 31\_Custom Exceptions

Custom exceptions in C# allow developers to define their own exception types when none of the existing exceptions in the .NET framework adequately describe the problem being encountered. Custom exceptions are particularly useful in scenarios where you want to provide more meaningful error messages or handle specific application scenarios that are not covered by standard exceptions.

### **When to Create a Custom Exception**

You typically create a custom exception when:

* The existing .NET exceptions do not accurately represent the error condition in your application.
* You need to provide a specific error message or encapsulate additional error details that aren't covered by standard exceptions.

### **Example Scenario**

Consider an ASP.NET web application where a user should only have one active session. If the user tries to log in from a second browser window while already logged in, the application should throw a custom exception stating that the user is already logged in from another window.

### **Steps to Create a Custom Exception**

1. **Create a Class Derived from System.Exception**:
   * Your custom exception class should inherit from the System.Exception class.
   * As a convention, the class name should end with the suffix Exception.
2. **Define Constructors**:
   * Provide a public constructor that accepts a string parameter (the error message).
   * Pass this parameter to the base class constructor.
   * Optionally, you can also overload the constructor to include inner exceptions for tracking back the original exception.
3. **Serialization Support (Optional)**:
   * If you want your custom exception to be serializable, mark it with the [Serializable] attribute.
   * Provide a constructor that takes SerializationInfo and StreamingContext parameters and calls the base class constructor that takes these parameters.

### 

### 

### **Example Code**

Here's an example of a custom exception that handles the scenario where a user tries to log in from multiple browser windows:

using System;

[Serializable]

public class UserAlreadyLoggedInException : Exception

{

// Default constructor

public UserAlreadyLoggedInException()

: base("User is already logged in from another session.")

{

}

// Constructor with custom message

public UserAlreadyLoggedInException(string message)

: base(message)

{

}

// Constructor with custom message and inner exception

public UserAlreadyLoggedInException(string message, Exception innerException)

: base(message, innerException)

{

}

// Constructor for serialization support

protected UserAlreadyLoggedInException(System.Runtime.Serialization.SerializationInfo info,

System.Runtime.Serialization.StreamingContext context)

: base(info, context)

{

}

}

### **Usage of the Custom Exception**

Here's how you can use the UserAlreadyLoggedInException in your application:

public class AuthService

{

private static Dictionary<string, bool> activeSessions = new Dictionary<string, bool>();

public void Login(string username)

{

if (activeSessions.ContainsKey(username) && activeSessions[username])

{

throw new UserAlreadyLoggedInException($"User '{username}' is already logged in from another session.");

}

// Simulate user login

activeSessions[username] = true;

Console.WriteLine($"{username} logged in successfully.");

}

public void Logout(string username)

{

if (activeSessions.ContainsKey(username))

{

activeSessions[username] = false;

Console.WriteLine($"{username} logged out successfully.");

}

}

}

class Program

{

static void Main()

{

AuthService authService = new AuthService();

try

{

authService.Login("user1");

authService.Login("user1"); // This will throw the custom exception

}

catch (UserAlreadyLoggedInException ex)

{

Console.WriteLine(ex.Message);

}

}

}

### **Explanation**

1. **Custom Exception Class**:
   * UserAlreadyLoggedInException is defined to represent a specific error condition where a user attempts to log in multiple times.
   * It includes several constructors, allowing it to be used in various scenarios, such as with or without an inner exception or a custom message.
2. **Using the Custom Exception**:
   * In the AuthService class, the Login method checks if the user is already logged in by checking the activeSessions dictionary.
   * If the user is already logged in, it throws a UserAlreadyLoggedInException.
   * The Main method demonstrates how to catch and handle this custom exception.

### **Summary**

* **Custom exceptions** are powerful tools for handling specific error conditions that aren't adequately described by the existing .NET exceptions.
* **Serialization** support in custom exceptions ensures that they can be used across application domains or in remoting scenarios.
* Custom exceptions improve the clarity and maintainability of your code by providing specific, meaningful error messages tailored to your application's needs.

# 32\_Access Modifier

### **Types vs. Type Members**

In C#, types are the fundamental building blocks of a program, and they define the structure and behavior of objects. **Type members** are the components that reside within these types and provide the functionality that the type offers.

* **Types**: These include classes, structs, enums, interfaces, and delegates. Types define the structure (fields, properties) and behavior (methods) of objects.
* **Type Members**: These include fields, properties, constructors, methods, events, indexers, operators, etc., that reside within a type.

### **Access Modifiers in C#**

C# provides five access modifiers to control the visibility and accessibility of types and their members. However, it's important to note that **types** can only use **public** and **internal** modifiers, while **type members** can use all five.

#### **1. Private**

* **Type Members**: Accessible only within the containing type (class, struct).
* **Usage**: Use this when you want to hide the member from all outside classes, including derived classes.

#### **2. Protected**

* **Type Members**: Accessible within the containing type and any derived types.
* **Usage**: Use this when you want to allow access to the member in derived classes, but not outside the inheritance hierarchy.

#### **3. Internal**

* **Types & Type Members**: Accessible anywhere within the same assembly.
* **Usage**: Use this to limit access to other types or members within the same assembly (project).

#### **4. Protected Internal**

* **Type Members**: Accessible within the same assembly and from derived classes in other assemblies.
* **Usage**: This is a combination of protected and internal. Use this when you want the member to be accessible in derived classes outside the assembly, but still within the same assembly.

#### **5. Public**

* **Types & Type Members**: Accessible from anywhere, with no restrictions.
* **Usage**: Use this when you want the type or member to be available to any other code.

### **Example to Illustrate Access Modifiers**

Let's take a look at a simple example to demonstrate these concepts:

public class Customer // Public type, accessible anywhere

{

private int customerId; // Private member, accessible only within Customer class

protected string customerName; // Protected member, accessible within Customer class and its derived classes

internal string customerAddress; // Internal member, accessible anywhere within the same assembly

protected internal string customerPhoneNumber; // Protected Internal member, accessible within the same assembly and derived classes outside the assembly

public string CustomerEmail { get; set; } // Public member, accessible anywhere

public void DisplayDetails() // Public method, accessible anywhere

{

Console.WriteLine($"Customer ID: {customerId}, Name: {customerName}, Address: {customerAddress}, Phone: {customerPhoneNumber}, Email: {CustomerEmail}");

}

}

public class VIPCustomer : Customer // Derived class of Customer

{

public void ShowVIPDetails()

{

// customerId is not accessible because it's private

Console.WriteLine($"Name: {customerName}"); // Accessible because it's protected

Console.WriteLine($"Address: {customerAddress}"); // Accessible because it's internal

Console.WriteLine($"Phone: {customerPhoneNumber}"); // Accessible because it's protected internal

}

}

internal class Order // Internal type, accessible only within the same assembly

{

public void ShowOrderDetails()

{

Console.WriteLine("Order details are shown here.");

}

}

### **Summary**

* **Types** (like Customer and Order) can only be public or internal.
* **Type Members** (like fields, properties, and methods) can be private, protected, internal, protected internal, or public.
* **Private** members are accessible only within the containing class, **protected** members are accessible within derived classes, **internal** members are accessible within the same assembly, and **protected internal** members combine the rules of protected and internal.
* **Public** members and types are accessible from anywhere.

# 33\_Enums

### **Why Use Enums in C#?**

Enums, or enumerations, are a way to define a set of named integral constants. They make the code more readable and maintainable by replacing magic numbers (hard-coded integers) with meaningful names.

#### **Benefits of Enums**

1. **Readability**:
   * Enums give meaningful names to integral constants, making the code easier to read and understand.
   * For example, instead of using numbers like 1 or 2, you can use EnumType.OptionA or EnumType.OptionB, which makes the intent clear.
2. **Maintainability**:
   * By using enums, you centralize the definitions of related constants in one place. This makes it easier to manage and update the code if the values or the meaning of the constants change.

#### **Characteristics of Enums in C#**

1. **Strongly Typed Constants**:
   * Enums in C# are strongly typed, which means that values of an enum type are not interchangeable with other integral types without an explicit cast. This prevents accidental misuse of constants that aren't meant to be used interchangeably.
2. **Underlying Type**:
   * The default underlying type for enum elements is int. However, you can change this to other integral types like byte, short, long, etc., if needed.
3. **Default Value**:
   * The default value for the first element in an enum is 0, and subsequent values are incremented by 1 unless explicitly defined.
4. **Customization**:
   * You can customize the underlying values of enum elements if you need specific values.
5. **Value Type**:
   * Enums are value types, which means they are stored on the stack and passed by value.
6. **Enum Keyword vs. Enum Class**:
   * The enum keyword is used to define enumerations.
   * The Enum class in C# provides static methods like GetValues() and GetNames() to retrieve the values and names of the enumeration.

### **Example of Using Enums in C#**

Consider the following example where we use an enum to represent days of the week:

public enum Weekday

{

Sunday, // 0

Monday, // 1

Tuesday, // 2

Wednesday, // 3

Thursday, // 4

Friday, // 5

Saturday // 6

}

public class Program

{

static void Main()

{

Weekday today = Weekday.Monday;

// Output: Today is Monday

Console.WriteLine($"Today is {today}");

// Checking enum value

if (today == Weekday.Monday)

{

Console.WriteLine("Start of the workweek!");

}

// Getting enum integer value

int dayValue = (int)today;

Console.WriteLine($"Numeric value of {today} is {dayValue}");

}

}

### 

### 

### **Customizing Enum Values**

You can also define custom values for enum members:

public enum ErrorCode

{

None = 0,

NotFound = 404,

ServerError = 500

}

public class Program

{

static void Main()

{

ErrorCode code = ErrorCode.NotFound;

// Output: Error Code: 404

Console.WriteLine($"Error Code: {(int)code}");

}

}

### **Conclusion**

Enums in C# are powerful tools that improve code readability and maintainability by providing meaningful names for sets of related constants. They enforce type safety, prevent errors, and make the code more intuitive to understand and work with.