1. Basic Structure of a C# Program

csharp Copy code

```csharp
using System; // Namespace

class Program
{
    static void Main(string[] args) // Entry point of the program
    {
        Console.WriteLine("Hello, World!"); // Output statement
    }
}
```

- **using System;**: Includes the System namespace.
- **class Program**: Defines a class named `Program`.
- **static void Main()**: The entry point for the C# program where execution starts.

---

2. Variables & Data Types

C# is statically typed, meaning you must declare the type of variable before use.

Common Data Types:

- `int` (Integer): Holds whole numbers.
- `double` (Double precision float): Holds decimal numbers.
- `string`: Holds text.
- `bool`: Holds true/false values.

csharp Copy code

```csharp
int num = 10;
double price = 29.99;
string name = "John";
bool isActive = true;
```

Advanced Data Types:

- **Arrays**: Collection of items of the same type.
- **Lists**: Dynamic array using `List<T>`.
- **Dictionaries**: Key-value pairs.

csharp Copy code

```csharp
int[] numbers = { 1, 2, 3, 4 };
List<string> fruits = new List<string> { "Apple", "Banana" };
Dictionary<string, int> ageMap = new Dictionary<string, int> { {"John", 30}, {"Jane",
```

3. Control Structures

If-Else Statements:

csharp Copy code

```csharp
if (num > 5)
{
    Console.WriteLine("Greater than 5");
}
else
{
    Console.WriteLine("5 or less");
}
```

Switch Statement:

csharp Copy code

```csharp
int day = 2;
switch (day)
{
    case 1:
        Console.WriteLine("Monday");
        break;
    case 2:
        Console.WriteLine("Tuesday");
        break;
    default:
        Console.WriteLine("Unknown Day");
        break;
}
```

Loops:

- **For Loop**:

csharp Copy code

```csharp
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

- **While Loop**:

csharp Copy code

```csharp
int i = 0;
while (i < 5)
{
    Console.WriteLine(i);
    i++;
}
```

4. Object-Oriented Programming (OOP) Concepts

4.1 Classes and Objects

Classes are blueprints, and objects are instances of those classes.

csharp Copy code

```csharp
class Car
{
    public string Brand;
    public string Model;

    public void Start()
    {
        Console.WriteLine("Car started");
    }
}

Car myCar = new Car();
myCar.Brand = "Toyota";
myCar.Model = "Corolla";
myCar.Start();
```

4.2 Encapsulation

Encapsulation hides the internal state of an object using **private** access modifier and exposes the functionality through **public** methods or properties.

csharp Copy code

```csharp
class Person
{
    private string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}
```

4.3 Inheritance

Inheritance allows one class to inherit fields and methods from another class.

csharp Copy code

```csharp
class Animal
{
    public void Eat()
    {
        Console.WriteLine("Eating...");
    }
}

class Dog : Animal
{
    public void Bark()
    {
        Console.WriteLine("Barking...");
    }
}

Dog myDog = new Dog();
myDog.Eat();  // Inherited method
myDog.Bark(); // Dog's own method
```

4.4 Polymorphism

Polymorphism means the ability to process objects differently based on their data type or class.

- **Method Overriding** (Run-time Polymorphism):

csharp Copy code

```csharp
class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal sound");
    }
}

class Cat : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Meow");
    }
}

Animal myCat = new Cat();
myCat.Speak(); // Output: Meow (Cat version is called)
```

- **Method Overloading** (Compile-time Polymorphism):

csharp Copy code

```csharp
class MathOperations
{
    public int Add(int a, int b) => a + b;
    public double Add(double a, double b) => a + b;
}

MathOperations math = new MathOperations();
Console.WriteLine(math.Add(5, 6));     // Output: 11
Console.WriteLine(math.Add(5.5, 6.5)); // Output: 12.0
```

4.5 Abstraction

Abstraction focuses on hiding implementation details and showing only the essential features of an object.

- **Abstract Class**:

csharp Copy code

```csharp
abstract class Shape
{
    public abstract void Draw();
}

class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing Circle");
    }
}
```

- **Interface**: Used for full abstraction.

csharp Copy code

```csharp
interface IShape
{
    void Draw();
}

class Rectangle : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing Rectangle");
    }
}
```

7. Access Modifiers in C#

Access modifiers define the visibility and accessibility of classes, methods, and variables.

7.1 Types of Access Modifiers:

- **public**: Accessible from any other class.
- **private**: Accessible only within the same class.
- **protected**: Accessible within the same class and by derived classes.
- **internal**: Accessible only within the same assembly.
- **protected internal**: Accessible within the same assembly or from derived classes.
- **private protected**: Accessible only within the same class or derived classes in the same assembly.

Example:

csharp Copy code

```csharp
class Car
{
    public string Brand;         // Accessible from anywhere
    private string Model;        // Accessible only within the Car class
    protected int Year;          // Accessible within Car class and derived classes
    internal string Color;       // Accessible within the same assembly
    protected internal string EngineType; // Accessible within the assembly and der
}
```

Default Access Modifier:

- The default access modifier for class members is **private**, meaning that without explicitly specifying an access modifier, the member is considered private.

---

8. Methods in C#

8.1 Basic Method Structure:

A method in C# is a block of code that performs a specific task. Methods can return values or be `void` if they don't return anything.

csharp Copy code

```csharp
class MathOperations
{
    public int Add(int a, int b)  // Method that returns the sum of two integers
    {
        return a + b;
    }

    public void PrintMessage()  // Void method that prints a message
    {
        Console.WriteLine("Hello, World!");
    }
}
```

8.2 Method Parameters:

C# methods can take parameters in various ways: value parameters, `ref`, `out`, and `params`.

---

9. Parameter Types

9.1 Value Parameters (Default Behavior)

- By default, parameters are passed by value, meaning that any changes to the parameter inside the method do not affect the original argument.

csharp Copy code

```csharp
public void Increment(int number)
{
    number++;  // This change will not affect the original argument
}

int num = 10;
Increment(num);
Console.WriteLine(num); // Output: 10
```

9.2 `ref` Keyword (Pass by Reference)

- With `ref`, you pass the argument by reference, meaning changes inside the method affect the original argument. The variable must be initialized before being passed.

csharp Copy code

```csharp
public void Increment(ref int number)
{
    number++;  // This change will affect the original argument
}

int num = 10;
Increment(ref num);
Console.WriteLine(num); // Output: 11
```

9.3 `out` Keyword (Output Parameters)

- Similar to `ref`, but the variable does not need to be initialized before passing it. It **must** be assigned within the method.

csharp Copy code

```csharp
public void GetValues(out int x, out int y)
{
    x = 5;  // Must assign values to x and y inside the method
    y = 10;
}

int a, b;
GetValues(out a, out b);
Console.WriteLine(a + ", " + b); // Output: 5, 10
```

9.4 `params` Keyword (Variable Number of Parameters)

- `params` allows a method to accept a variable number of arguments of a specified type.

csharp Copy code

```csharp
public void PrintNumbers(params int[] numbers)
{
    foreach (int num in numbers)
    {
        Console.WriteLine(num);
    }
}

PrintNumbers(1, 2, 3, 4, 5); // Output: 1, 2, 3, 4, 5
```

---

10. Properties in C#

Properties are a way to control the accessibility of class fields. They can be used to validate data before allowing access.

Auto-Implemented Properties:

csharp Copy code

```csharp
class Person
{
    public string Name { get; set; }  // Automatically implements a private field
    public int Age { get; set; }
}
```

Properties with Logic:

csharp Copy code

```csharp
class Person
{
    private int age;

    public int Age
    {
        get { return age; }
        set
        {
            if (value >= 0)
                age = value;
        }
    }
}
```

11. Static Members

11.1 Static Fields and Methods

- `static` fields and methods belong to the class itself, rather than to a specific instance of the class.

csharp Copy code

```csharp
class Calculator
{
    public static int Multiply(int a, int b)  // Static method
    {
        return a * b;
    }
}

int result = Calculator.Multiply(5, 3); // Called without creating an instance
Console.WriteLine(result); // Output: 15
```

---

12. Constructors in C#

Constructors are special methods that are automatically called when an object is created. A class can have multiple constructors (constructor overloading).

Default Constructor:

csharp Copy code

```csharp
class Car
{
    public string Brand;

    // Default constructor
    public Car()
    {
        Brand = "Toyota";
    }
}

Car myCar = new Car();
Console.WriteLine(myCar.Brand); // Output: Toyota
```

Parameterized Constructor:

csharp Copy code

```csharp
class Car
{
    public string Brand;
    public string Model;

    // Parameterized constructor
    public Car(string brand, string model)
    {
        Brand = brand;
        Model = model;
    }
}

Car myCar = new Car("Honda", "Civic");
Console.WriteLine(myCar.Brand + " " + myCar.Model); // Output: Honda Civic
```

Static Constructor:

- A static constructor initializes static members of a class.

csharp Copy code

```csharp
class Configuration
{
    public static string ConfigName;

    static Configuration()  // Static constructor
    {
        ConfigName = "DefaultConfig";
    }
}
```

13. Indexers in C#

Indexers allow objects to be indexed like arrays. They are commonly used to provide array-like access to class members.

csharp Copy code

```csharp
class Team
{
    private string[] players = new string[5];

    public string this[int i]
    {
        get { return players[i]; }
        set { players[i] = value; }
    }
}

Team team = new Team();
team[0] = "John";
Console.WriteLine(team[0]); // Output: John
```

---

14. Namespaces

Namespaces are used to organize code into logical groups and to avoid naming conflicts.

csharp Copy code

```csharp
namespace MyApp.Utilities
{
    class Calculator
    {
        public int Add(int a, int b)
        {
            return a + b;
        }
    }
}
```

To use the `Calculator` class:

csharp Copy code

```csharp
using MyApp.Utilities;
```

13. Indexers in C#

Indexers allow objects to be indexed like arrays. They are commonly used to provide array-like access to class members.

csharp Copy code

```csharp
class Team
{
    private string[] players = new string[5];

    public string this[int i]
    {
        get { return players[i]; }
        set { players[i] = value; }
    }
}

Team team = new Team();
team[0] = "John";
Console.WriteLine(team[0]); // Output: John
```

---

14. Namespaces

Namespaces are used to organize code into logical groups and to avoid naming conflicts.

csharp Copy code

```csharp
namespace MyApp.Utilities
{
    class Calculator
    {
        public int Add(int a, int b)
        {
            return a + b;
        }
    }
}
```

To use the `Calculator` class:

csharp Copy code

```csharp
using MyApp.Utilities;
```

15. Enums in C#

Enums are value types that allow you to define a group of named constants.

csharp Copy code

```csharp
enum DaysOfWeek { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }

DaysOfWeek today = DaysOfWeek.Monday;
Console.WriteLine(today);  // Output: Monday
```

---

16. Structs in C#

Structs are value types, unlike classes which are reference types.

csharp Copy code

```csharp
struct Point
{
    public int X;
    public int Y;

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }
}

Point point = new Point(10, 20);
Console.WriteLine(point.X + ", " + point.Y);  // Output: 10, 20
```

---

17. Nullable Types

C# allows you to use nullable types with value types like `int`, `bool`, etc.

csharp Copy code

```csharp
int? num = null;

if (num.HasValue)
{
    Console.WriteLine(num.Value);
}
else
{
    Console.WriteLine("No value");
}
```