Detailed Explanation of Methods in C#

- **A method in C# is a block of code that performs a specific task.**
- **It helps in code organization, reusability, and reducing redundancy.**
- **Methods can take parameters, perform operations, and return values.**

Defining a Method in C#

- **A method consists of several key parts:**
- **Return Type: Specifies the type of data the method returns.**
- **If it does not return anything, we use void.**
- **Method Name: Should be unique and descriptive of its functionality.**
- **Parameters (Optional): Values that can be passed to the method for processing.**
- **Method Body: Contains the instructions that execute when the method is called.**
- **Return Statement (Optional): Used to return a value if the return type is not void.**

🔷 Basic Syntax of a Method

csharp Copy Edit

```csharp
returnType MethodName(parameters)
{
    // Method body (code execution)
    return value; // Optional (if returnType is not void)
}
```

Types of Methods in C#

- **Methods with a Return Value**

  Returns a specific type like int, string, bool, etc.

- **Void Methods (No Return Value)**

  Performs an action but does not return anything.

- **Methods with Parameters**

  Accepts input values (parameters) for processing inside the method.

- **Methods without Parameters**

  Does not require input values, just executes an operation.

Static Methods vs. Instance Methods

1- Static Methods

- **Can be called without creating an object (object) of the class.**
- **Belong to the class itself rather than objects.**
-

csharp Copy Edit

```csharp
class MathOperations
{
    public static int Square(int number)
    {
        return number * number;
    }
}

class Program
{
    static void Main()
    {
        int result = MathOperations.Square(4);
        Console.WriteLine("Square: " + result);
    }
}
```

Instance Methods

- **Require creating an object of the class before calling the method.**
- **Belong to objects rather than the class itself.**

csharp Copy Edit

```csharp
class Person
{
    public string Name;

    public void Introduce()
    {
        Console.WriteLine("Hello, my name is " + Name);
    }
}

class Program
{
    static void Main()
    {
        Person person = new Person();
        person.Name = "Ahmed";
        person.Introduce();
    }
```

🔷 Methods with Default Parameter Values

You can define **default values** for parameters so that the method can be called without providing all arguments.

csharp Copy Edit

```csharp
static void PrintInfo(string name, int age = 25)
{
    Console.WriteLine($"Name: {name}, Age: {age}");
}

static void Main()
{
    PrintInfo("Ali"); // Default age will be used
    PrintInfo("Sara", 30);
}
```

🔷 Methods Returning Multiple Values (Tuples)

A method can return **multiple values** using **tuples**.

csharp Copy Edit

```csharp
static (int, int) GetMinMax(int a, int b)
{
    return (Math.Min(a, b), Math.Max(a, b));
}

static void Main()
{
    var (min, max) = GetMinMax(5, 10);
    Console.WriteLine($"Min: {min}, Max: {max}");
```

Using => (Arrow Expression) in C#

- **In C#, the => (Arrow Expression) is used to simplify method and property definitions.**
- **It is known as an "Expression-bodied Member" and helps make the code more concise.**
- **Using => in Methods**
- **You can use => instead of {} when the method contains only a single expression.**

🔷 Regular Method (Without `=>`)

csharp Copy Edit

```csharp
static int Square(int x)
{
    return x * x;
}
```

🔷 Using `=>`

csharp Copy Edit

```csharp
static int Square(int x) => x * x;
```

- The `=>` automatically returns the result without needing `{}` or `return`.

When to Use =>?

- **When the method has only one expression.**
- **To make code shorter and more readable.**
- **When defining simple properties in objects.**

Pass by Value vs. Pass by Reference in C#

- **In C#, when passing arguments to methods, you can do it in two ways:**
- **Pass by Value (default behavior)**
- **Pass by Reference (ref, out, in)**

1. Pass by Value (Default Behavior)

- **When passing a variable by value, a copy of the variable is sent to the method.**
- **Changes inside the method do not affect the original variable.**

Default behavior for value types (int, double, char, bool, etc.).

```
class Program
{
    static void Increment(int num)
    {
        num++; // This changes the local copy, not the original value
    }

    static void Main()
    {
        int number = 10;
        Increment(number);
        Console.WriteLine("Number after method call: " + number); // Still 10
    }
}
```

Copy Edit

🔷 Output:

```
pgsql

Number after method call: 10
```

Copy Edit

✅ The original value **remains unchanged** because `num` is a copy of `number`.

2. Pass by Reference (ref, out, in)

- **When passing a variable by reference, the method receives a reference to the original variable, not a copy.**
- **Changes inside the method affect the original variable.**
- **Used with the ref, out, or in keywords.**

```
class Program
{
    static void Increment(ref int num)
    {
        num++; // This changes the original value
    }

    static void Main()
    {
        int number = 10;
        Increment(ref number);
        Console.WriteLine("Number after method call: " + number); // Now 11
    }
}
```

Copy Edit

🔷 Output:

```
pgsql

Number after method call: 11
```

Copy Edit

✅ `ref` allows the method to modify the **original variable**.

out (Pass by Reference, Must Assign New Value)

- **The variable does not need to be initialized before passing it.**
- **The method must assign a value before it exits.**
- **Used when a method needs to return multiple values.**

Copy Edit

```
class Program
{
    static void GetValues(out int x, out int y)
    {
        x = 5;  // Must assign a value
        y = 10; // Must assign a value
    }

    static void Main()
    {
        int a, b; // No need to initialize
        GetValues(out a, out b);
        Console.WriteLine($"a = {a}, b = {b}"); // a = 5, b = 10
    }
}
```

🔷 Output:

Copy Edit

```
ini

a = 5, b = 10
```

✅ `out` ensures that the method **assigns values** be[illegible]e exiting.

in (Pass by Reference, Read-Only)

- **The method cannot modify the variable (read-only).**
- **Used for performance optimization (especially with large objects).**

```
class Program
{
    static void PrintValue(in int num)
    {
        Console.WriteLine("Value: " + num);
        // num++; // ❌ Error: Cannot modify 'num' because it is passed as 'in'
    }

    static void Main()
    {
        int number = 100;
        PrintValue(in number);
    }
}
```

Copy Edit

🔷 Output:

```
makefile

Value: 100
```

Copy Edit

✅ `in` prevents modification inside the method.

🔷 Comparison Table:

Feature	Pass by Value	`ref` (Pass by Reference)	`out` (Pass by Reference)	`in` (Read-Only Reference)
Requires Initialization?	✅ Yes	✅ Yes	❌ No	✅ Yes
Can be Modified in Method?	❌ No	✅ Yes	✅ Yes (must assign new value)	❌ No
Must Assign a New Value?	❌ No	❌ No	✅ Yes	❌ No
Use Case	Default behavior	Modify existing value	Return multiple values	Performance optimization

Summary

1. **Pass by Value: The method gets a copy, and changes do not affect the original variable.**
2. **Pass by Reference (ref): The method modifies the original variable.**
3. **Pass by Reference (out): Used when the method must assign a new value.**
4. **Pass by Reference (in): The variable is read-only, preventing modification.**