1. Pass By Value vs Pass By Reference

Pass By Value (Default Behavior)

When you pass arguments to methods in C#, by default, they are passed **by value**. This means:

- A **copy** of the variable's value is created and passed to the method
- Any modifications made to the parameter inside the method **do NOT affect** the original variable
- For value types (int, double, struct, etc.), the actual value is copied
- For reference types (arrays, classes), the reference address is copied, but it still points to the same object in memory

Example from the code (commented out):

```
public void swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
// Calling: op.swap(a, b);
// Result: a and b remain unchanged outside the method
```

Why it doesn't work: The parameters `x` and `y` are copies of `a` and `b`. Swapping the copies doesn't affect the originals.

Pass By Reference (using `ref` keyword)

The `ref` keyword allows you to pass a variable **by reference**, meaning:

- The method receives a **reference** to the original variable, not a copy
- Any modifications made inside the method **directly affect** the original variable
- Both value types and reference types can be passed by reference
- The variable **must be initialized** before being passed as a `ref` parameter
- You must use the `ref` keyword at both the method declaration AND the method call

Example from the code:

```
public void swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
// Calling: op.swap(ref a, ref b);
// Result: a and b are successfully swapped
```

Key Points:

- `ref` creates an **alias** to the original variable
- Memory efficient for large structures
- Allows methods to modify the caller's variables directly

Arrays and Pass By Reference

Arrays are reference types, but when you pass an array to a method normally, you're passing a copy of the reference. This means:

- You can modify the **contents** of the array (individual elements)
- You **cannot** reassign the entire array reference

Example from the code:

```
public void increase(int[] x)
{
    for (int i = 0; i < x.Length; i++)
        x[i]++;
}
// This works: modifies array contents
```

But to swap entire array references, you need `ref`:

```
public void swap(ref int[] x, ref int[] y)
{
    int[] temp = x;
    x = y;
    y = temp;
}
// Now you can swap which arrays the variables point to
```

---

2. The `out` Keyword

The `out` keyword is similar to `ref`, but with important differences:

Characteristics of `out`:

- Used to **return multiple values** from a method
- The variable does **NOT need to be initialized** before passing to the method
- The method **MUST assign** a value to the `out` parameter before returning
- Like `ref`, you must use the `out` keyword at both declaration and call site

Example from the code:

```
public int calc(int x, int y, out int mul, out int sub, in int z)
{
    mul = x * y;    // Must assign value
    sub = x - y;    // Must assign value
    return x + y;
}

// Calling:
int s, m, sub;
s = op.calc(7, 3, out m, out sub);
// Now: s=10, m=21, sub=4
```

When to use `out`:

- When a method needs to return more than one value
- When you want to clearly indicate that a parameter is for output only
- Common in TryParse methods: `int.TryParse("123", out int result)`

`ref` vs `out` Comparison:

Feature	`ref`	`out`
Must initialize before passing	✓ Yes	✗ No
Must assign in method	✗ No	✓ Yes
Can read initial value	✓ Yes	✗ No (unreliable)
Purpose	Two-way communication	Return multiple values

---

3. The `in` Keyword

The `in` keyword was introduced in C# 7.2 for **read-only references**.

Characteristics of `in`:

- Passes arguments **by reference** (avoiding copying)
- Guarantees the method **cannot modify** the parameter
- Useful for **performance optimization** with large structs
- The compiler enforces immutability

Example from the code:

```
public int calc(int x, int y, out int mul, out int sub, in int z)
{
    // z can be read but not modified
    // Attempting to modify z would cause a compile error
    return x + y;
}
```

When to use `in`:

- Passing large structs to avoid copying overhead
- When you want to guarantee a parameter won't be modified
- For performance-critical code where struct copying is expensive

---

4. Method Overloading

Method overloading allows you to define multiple methods with the **same name** but **different parameters**.

Rules for Overloading:

- Methods must have different **parameter lists** (number, type, or order of parameters)
- Return type alone is **NOT sufficient** for overloading
- Parameter names don't matter, only types matter

Examples from the code:

```
// Overloading the swap method
public void swap(ref int x, ref int y) { ... }
```

```
public void swap(ref int[] x, ref int[] y) { ... }

// Different parameter types allow overloading
```

Why Overloading is Useful:

- Provides intuitive API with the same method name for similar operations
- Compiler chooses the correct method based on arguments passed
- Makes code more readable and maintainable

---

5. Default Parameters (Optional Parameters)

C# allows you to specify **default values** for method parameters, making them optional when calling the method.

Characteristics:

- Default parameters must appear **after all required parameters**
- You can specify any constant value as a default
- Caller can omit optional parameters

Example from the code (commented):

```
public int sum(int x, int y, int z=0, int a=0)
{
    return x + y + z + a;
}

// Can be called as:
op.sum(1, 2);        // z=0, a=0 (uses defaults)
op.sum(1, 2, 3);     // a=0 (uses default)
op.sum(1, 2, 3, 4);  // all parameters specified
```

Constructor with Default Parameters:

```
public employee(int _id=0, string _name="", int _age=6)
{
    id = _id;
    name = _name;
    age = _age;
}
```

```
// Flexible instantiation:
employee em1 = new employee(1, "ali", 22);    // All specified
employee em2 = new employee();                 // All defaults
employee em3 = new employee(1);                // Partial
```

Best Practices:

- Use defaults for parameters that have a "sensible" default value
- Don't overuse defaults as they can make code less clear
- Consider using method overloading for complex scenarios

---

6. Named Parameters

Named parameters allow you to specify arguments by **parameter name** rather than position.

Characteristics:

- Improves code readability
- Allows you to skip parameters (when combined with defaults)
- Order of named parameters doesn't matter
- Can mix positional and named, but positional must come first

Examples from the code:

```
employee em5 = new employee(_name:"ali");              // Skip _id, _age
employee em6 = new employee(_age:6);                    // Skip _id, _name
employee em7 = new employee(_age:6, _name:"motafa");   // Different order
employee em8 = new employee(_age:5, _id:4);            // Skip _name
employee em9 = new employee(_name:"ali", _id:4);       // Skip _age
```

When to Use Named Parameters:

- When calling methods with many parameters
- When using optional parameters and you want to skip some
- To make code self-documenting and more readable
- When parameter order is not intuitive

Mixed Usage:

```
// Valid: positional then named
employee em = new employee(1, _name:"ali");

// Invalid: named then positional
// employee em = new employee(_id:1, "ali"); // Compile error!
```

---

7. The `params` Keyword

The `params` keyword allows a method to accept a **variable number of arguments** of the same type.

Characteristics:

- Must be the **last parameter** in the method signature
- Can only have **one** `params` parameter per method
- Internally treated as an array
- Caller can pass individual values, an array, or no arguments

Example from the code:

```
public int sum(params int[] arr)
{
    int sum = 0;
    for (int i = 0; i < arr.Length; i++)
    {
        sum += arr[i];
    }
    return sum;
}

// Multiple ways to call:
op.sum(1, 2, 3);                          // Individual arguments
op.sum(1, 2, 3, 4, 5, 33, 44, 55);        // Many arguments
op.sum(new int[] { 1, 2, 3, 4, 5 });      // Array
op.sum();                                   // No arguments (empty array)
```

Common Uses:

- `string.Format()` and `Console.WriteLine()` use `params`
- When you don't know how many arguments will be passed
- For cleaner API design (avoiding array creation in caller code)

Advantages:

- **Flexibility**: Caller doesn't need to create an array explicitly
- **Readability**: Makes method calls cleaner
- **Convenience**: Simplifies common patterns

---

8. Structs vs Classes

The code defines `operation` as a `struct` and `employee` as a `class`. Let's understand the differences:

Struct Characteristics:

- **Value type** (stored on the stack)
- Copied when assigned or passed to methods
- Cannot inherit from other structs or classes
- Cannot be null (unless made nullable: `Nullable<T>` or `T?`)
- Better performance for small, simple data structures
- Automatically inherits from `System.ValueType`

Class Characteristics:

- **Reference type** (stored on the heap)
- Reference is copied when assigned (both variables point to same object)
- Supports inheritance
- Can be null
- Used for complex objects with behavior

When to use struct:

- Small data structures (generally < 16 bytes)
- Immutable objects
- Value semantics are desired
- Examples: Point, Color, Date

When to use class:

- Complex objects with behavior
- When you need inheritance
- When identity matters (two objects can be equal but distinct)

- Most object-oriented designs

---

9. Properties (Auto-Implemented)

The `employee` class uses auto-implemented properties:

```
public int id { get; set; }
public string name { get; set; }
public int age { get; set; }
```

What are Properties?

- Provide controlled access to private fields
- Can include logic in getters and setters
- Support encapsulation principle

Auto-Implemented Properties:

- Compiler automatically creates a hidden backing field
- Syntax is concise
- You can still add logic later without breaking code

Equivalent full syntax:

```
private int _id;
public int id
{
    get { return _id; }
    set { _id = value; }
}
```

Property Features:

- **Read-only**: Only `get` accessor
- **Write-only**: Only `set` accessor (rare)
- **Different access levels**: `public int age { get; private set; }`
- **Computed properties**: `public int BirthYear { get { return DateTime.Now.Year - age; } }`

---

10. Constructor Overloading

The `employee` class demonstrates constructor overloading:

```
public employee(int _id=0, string _name="", int _age=6)
{
    id = _id;
    name = _name;
    age = _age;
}
```

Constructor Characteristics:

- Special method that runs when an object is created
- Same name as the class
- No return type (not even `void`)
- Can be overloaded like regular methods

This Constructor Pattern:

- Uses default parameters for flexibility
- Allows multiple instantiation patterns with one constructor
- Alternative: Multiple explicit constructors

Traditional approach (multiple constructors):

```
public employee() : this(0, "", 6) { }
public employee(int _id) : this(_id, "", 6) { }
public employee(int _id, string _name) : this(_id, _name, 6) { }
public employee(int _id, string _name, int _age)
{
    id = _id;
    name = _name;
    age = _age;
}
```

---

11. Overriding `ToString()` Method

The `employee` class overrides the `ToString()` method:

```
public override string ToString()
{
    return $"{id}-{name}-{age}years old";
}
```

Why Override `ToString()`?

- `ToString()` is defined in `System.Object` (base of all types)
- Default implementation returns the type name
- Overriding provides meaningful string representation
- Automatically called by `Console.WriteLine()` and string interpolation

Usage:

```
employee em = new employee(1, "ali", 22);
Console.WriteLine(em);  // Calls ToString() automatically
// Output: 1-ali-22years old
```

Best Practices:

- Always override `ToString()` for better debugging
- Return human-readable representation
- Use string interpolation (`$""`) for clean syntax
- Keep it simple and informative

---

12. String Interpolation

The code uses string interpolation (C# 6.0+):

```
return $"{id}-{name}-{age}years old";
```

Advantages over Concatenation:

```
// Old way (concatenation)
return id + "-" + name + "-" + age + "years old";

// Old way (string.Format)
return string.Format("{0}-{1}-{2}years old", id, name, age);
```

```
// Modern way (interpolation)
return $"{id}-{name}-{age}years old";
```

Benefits:

- **Readability**: Code is clearer and more maintainable
- **Type safety**: Compile-time checking
- **Performance**: Compiler optimizes to efficient code
- **Flexibility**: Can include expressions: `$"Total: {x + y}"`

Advanced Features:

```
// Formatting
$"Price: {price:C2}"           // Currency with 2 decimals
$"Date: {date:yyyy-MM-dd}"     // Date formatting
$"Percent: {value:P}"          // Percentage

// Alignment
$"{name,20}"                   // Right-aligned in 20 chars
$"{name,-20}"                  // Left-aligned in 20 chars

// Expressions
$"Status: {age >= 18 ? "Adult" : "Minor"}"
```

---

13. Regions (`#region` directive)

The code uses `#region` preprocessor directives:

```
#region pass by value vs by ref
// code here
#endregion
```

Purpose:

- Organizes code into collapsible sections in IDEs
- Improves code navigation in large files
- Purely for developer convenience (no runtime effect)
- Helps with code reviews and maintenance

Best Practices:

- Use meaningful region names
- Don't overuse (prefer small, well-organized classes)
- Common regions: Fields, Properties, Constructors, Methods, Events
- Some developers prefer avoiding regions and using multiple files instead

---

Key Takeaways Summary

1. **Pass by Value vs Reference**: Default is by value (copies data), use `ref` to pass references
2. `out` **keyword**: For returning multiple values, doesn't require initialization
3. `in` **keyword**: Read-only reference parameter for performance
4. **Method Overloading**: Same name, different parameters
5. **Default Parameters**: Optional parameters with default values
6. **Named Parameters**: Call methods by parameter name for clarity
7. `params` **keyword**: Accept variable number of arguments
8. **Structs vs Classes**: Value types vs reference types
9. **Properties**: Encapsulate fields with controlled access
10. **Constructor Overloading**: Multiple ways to instantiate objects
11. `ToString()` **Override**: Provide meaningful string representation
12. **String Interpolation**: Modern, readable string formatting
13. **Regions**: Code organization tool for IDEs

These concepts are fundamental to writing clean, efficient, and maintainable C# code.

By Abdullah Ali

Contact : +201012613453