Complete Guide to C# Delegates

Table of Contents

What are Delegates?

Definition

A **delegate** is a type-safe function pointer. It's a reference type that holds a reference to a method with a matching signature.

Think of Delegates as:

- **C/C++**: Function pointers
- **C#**: Type-safe method references
- **Real World**: A phone contact that can call different people

Simple Analogy

```
Traditional Approach:
┌──────────────────────────────────────────┐
│ You want to do an operation              │
│ → Call specific method directly          │
│ → Hard-coded                             │
└──────────────────────────────────────────┘
```

```
With Delegates:
┌─────────────────────────────────────────┐
│ You want to do an operation             │
│ → Store method reference in delegate    │
│ → Call delegate (flexible!)             │
│ → Can change method at runtime          │
└─────────────────────────────────────────┘
```

Delegate Declaration & Usage

Step 1: Declare Delegate Type

```
// Delegate declaration
delegate int MyDel(int x, int y);
//       ↑   ↑     ↑
//       │   │     └── Parameters
//       │   └──────── Delegate name
//       └──────────── Return type

// This creates a new TYPE called MyDel
// Any method with signature: int MethodName(int, int) can be stored
```

Step 2: Create Methods that Match

```
class Operation
{
    // ✅ Matches MyDel signature
    public int Sum(int x, int y)
    {
        Console.WriteLine($"sum={x+y}");
        return x + y;
    }

    // ✅ Matches MyDel signature
    public int Sub(int x, int y)
    {
        Console.WriteLine($"sub={x-y}");
        return x - y;
    }

    // ✅ Matches MyDel signature (static is OK)
    public static int Div(int x, int y)
    {
        Console.WriteLine($"div={x/y}");
        return x / y;
    }

    // ❌ Does NOT match MyDel (wrong return type)
    public void Display(int x, int y) { }
}
```

Step 3: Create Delegate Instance

```
Operation op = new Operation();

// Method 1: Traditional (verbose)
MyDel d = new MyDel(op.Sum);

// Method 2: Simplified (C# 2.0+)
MyDel d = op.Sum;

// Method 3: Static method
MyDel d = Operation.Div;
```

Step 4: Invoke Delegate

```
// Method 1: Explicit Invoke
int result = d.Invoke(5, 3);

// Method 2: Shorthand (same as Invoke)
int result = d(5, 3);

Console.WriteLine(result);  // Output: sum=8
```

Complete Example

```
// 1. Declare delegate type
delegate int MyDel(int x, int y);

class Program
{
    static void Main()
    {
        Operation op = new Operation();
        // 2. Create delegate pointing to Sum
        MyDel d = op.Sum;

        // 3. Invoke
        int result = d(7, 3);  // Calls op.Sum(7, 3)
        Console.WriteLine(result);  // Output: sum=10

        // 4. Change method at runtime
        d = op.Sub;
        result = d(7, 3);  // Now calls op.Sub(7, 3)
        Console.WriteLine(result);  // Output: sub=4

        // 5. Use static method
        d = Operation.Div;
        result = d(8, 2);  // Calls Operation.Div(8, 2)
        Console.WriteLine(result);  // Output: div=4
    }
}
```

Delegate as Parameter

The Power of Delegates

```
// Instead of writing separate methods:
static void CalcSum(int x, int y) { /* ... */ }
static void CalcSub(int x, int y) { /* ... */ }
static void CalcMul(int x, int y) { /* ... */ }

// Write ONE flexible method:
static void Calc(int x, int y, MyDel operation)
{
    int result = operation(x, y);
    Console.WriteLine(result);
}
```

Usage Examples

```
Operation op = new Operation();

// Use different operations without changing Calc
Calc(7, 3, op.Sum);  // Output: sum=10
Calc(7, 3, op.Sub);  // Output: sub=4

Calc(7, 3, op.Mul);  // Output: mul=21

Calc(8, 2, Operation.Div);  // Output: div=4
```

Visual Flow

```
Calc(7, 3, op.Sum):
┌─────────────────────────────────────────┐
│ Calc Method Frame                       │
│  ┌─────────────────────────────────────┐│
│  │ x = 7                               ││
│  │ y = 3                               ││
│  │ operation = op.Sum ──────────┐      ││
│  └──────────────────────────────┼──────┘│
│                                 │       │
│                                 │       │
│ result = operation(x, y)        │       │
│        = op.Sum(7, 3) ←─────────┘       │
│        = 10                             │
└─────────────────────────────────────────┘
```

Multicast Delegates

Definition

A **multicast delegate** holds references to multiple methods. When invoked, it calls all methods in order.

Adding Methods

```
MyDel d = op.Mul;        // d points to Mul
d += op.Sum;             // d now points to Mul AND Sum
d += op.Sub;             // d now points to Mul, Sum, AND Sub

// Invoke :- calls all three methods in order!
int result = d(7, 3);
// Output:
// mul=21
// sum=10
// sub=4
// Returns: 4 (last method's return value)
```

Visual Representation

```
Single Delegate:
┌─────────────────┐
│   MyDel d       │
│   Target: op    │
│   Method: Sum   │
└─────────────────┘

       ↓
   op.Sum(x, y)


Multicast Delegate:
┌─────────────────────────────────────────┐
│   MyDel d (Invocation List)             │
│  ┌────────────────────────────────────┐ │
│  │ [0] Target: op, Method: Mul        │ │
│  │ [1] Target: op, Method: Sum        │ │
│  │ [2] Target: op, Method: Sub        │ │
│  └────────────────────────────────────┘ │
└─────────────────────────────────────────┘

       ↓           ↓           ↓
   op.Mul(x,y) op.Sum(x,y) op.Sub(x,y)
```

Complete Example

```
Operation op = new Operation();

// Start with mul
MyDel d = op.Mul;
Console.WriteLine(d(7, 3));  // Output: mul=21

Console.WriteLine("----------------");

// Add sum
d += op.Sum;
Console.WriteLine(d(7, 3));
// Output: mul=21
//         sum=10
//         10  ← Returns last method's result
```

```csharp
Console.WriteLine("----------------");

// Add sub
d += op.Sub;
Console.WriteLine(d(7, 3));
// Output: mul=21
//         sum=10
//         sub=4
//         4  ← Returns last method's result
```

Removing Methods

```csharp
MyDel d = op.Mul;
d += op.Sum;
d += op.Sub;
d += Operation.Div;

// Remove sub
d -= op.Sub;

d(8, 2);
// Output: mul=16
//         sum=10
//         div=4
// Note: sub is NOT called!
```

Combining Delegates

```csharp
MyDel d1 = op.Sum;
d1 += op.Sub;

MyDel d2 = op.Mul;

// Combine delegates
MyDel d3 = d1 + d2;
d3(7, 3);
// Output: sum=10
//         sub=4
//         mul=21

// Subtract delegates
MyDel d4 = d3 - d1;
```

```
d4(7, 3);
// Output: mul=21  ← Only mul remains
```

Important Notes

⚠ Multicast Delegate Return Values

- Only the **last method's** return value is returned
- Previous return values are discarded
- If you need all return values, use events or custom solution

```
MyDel d = op.Mul;       // Returns 21
d += op.Sum;            // Returns 10
d += op.Sub;            // Returns 4

int result = d(7, 3);
// result = 4  ← Only last one!
// Previous returns (21, 10) are lost!
```

Anonymous Methods

Definition

Anonymous methods are inline methods without a name, defined using the `delegate` keyword.

Syntax

```
// Named method approach:
class Operation
{
    public int Sum(int x, int y)
    {
        return x + y;
    }
}
MyDel d = op.Sum;
```

```
// Anonymous method approach:
MyDel d = delegate(int x, int y)
{
    return x + y;
};
```

Examples

Example 1: Simple Anonymous Method

```
MyDel d = delegate(int x, int y)
{
    return x - y;
};

Console.WriteLine(d(5, 3));  // Output: 2
```

Example 2: Anonymous Method as Parameter

```
// Instead of defining a separate method
Calc(4, 5, delegate(int x, int y)
{
    return x + y;
});
// Output: 9

// Another example
Calc(10, 3, delegate(int x, int y)
{
    return x * y;
});
// Output: 30
```

Example 3: Multi-line Anonymous Method

```
MyDel d = delegate(int x, int y)
{
    Console.WriteLine($"Computing: {x} and {y}");
    int result = x * y;
    Console.WriteLine($"Result: {result}");
    return result;
};
```

```
d(7, 3);
// Output: Computing: 7 and 3
//         Result: 21
//         21
```

When to Use

✓ **Use Anonymous Methods When:**

- Method is simple and used only once
- Don't want to clutter class with tiny methods
- Need closure over local variables

Avoid When:

- Method is complex
- Need to reuse the method
- Want better readability

Lambda Expressions

Definition

Lambda expressions are a more concise syntax for anonymous methods, using the `=>` operator.

Syntax Evolution

```
// 1. Named method
public int Add(int x, int y) { return x + y; }
MyDel d = Add;

// 2. Anonymous method
MyDel d = delegate(int x, int y) { return x + y; };

// 3. Lambda expression (full)
MyDel d = (int x, int y) => { return x + y; };

// 4. Lambda expression (type inference)
```

```
MyDel d = (x, y) => { return x + y; };

// 5. Lambda expression (expression body)
MyDel d = (x, y) => x + y;  // ← Most concise! ✨
```

Lambda Operator `=>`

```
(parameters) => expression/statement-block
     ↑               ↑
     │               └── What to do
     └────────────────── Input
```

Read as: "goes to" or "such that"

Examples

Example 1: Simple Lambda

```
MyDel d = (x, y) => x + y;
Console.WriteLine(d(5, 3));  // Output: 8

// Equivalent to:
MyDel d = delegate(int x, int y) { return x + y; };
```

Example 2: Lambda as Parameter

```
// Super concise!
Calc(4, 5, (x, y) => x + y);   // Output: 9
Calc(4, 5, (x, y) => x - y);   // Output: -1
Calc(4, 5, (x, y) => x * y);   // Output: 20
Calc(4, 5, (x, y) => x / y);   // Output: 0
```

Example 3: Multi-statement Lambda

```
MyDel d = (x, y) =>
{
    Console.WriteLine($"Input: {x}, {y}");
    int result = x * y;
    Console.WriteLine($"Output: {result}");
    return result;
};

d(7, 3);
```

```
// Output: Input: 7, 3
//         Output: 21
//         21
```

Example 4: Lambda with Collections

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Find all even numbers
List<int> evens = numbers.FindAll(n => n % 2 == 0);
// evens = [2, 4, 6, 8, 10]

// Double each number
List<int> doubled = numbers.Select(n => n * 2).ToList();
// doubled = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

// Sum of all numbers
int sum = numbers.Sum(n => n);  // sum = 55
```

Comparison Table

Feature	Anonymous Method	Lambda Expression
Syntax	`delegate(int x) { ... }`	`(x) => ...`
Type Inference	No	Yes
Conciseness	Verbose	Concise
Expression Body	No	Yes
Readability	Lower	Higher

Generic Delegates

Definition

Generic delegates use type parameters, making them reusable with different types.

Declaration

```
// Generic delegate
delegate T MyDel3<T, T1>(T1 x);
//       ↑        ↑  ↑    ↑
//       │        │  │    └── Parameter type
//       │        │  └─────── Type parameter 2
//       │        └────────── Type parameter 1
//       └─────────────────── Return type
```

Usage Examples

```
class Operation
{
    // int Display(string txt) - Returns T=int, Takes T1=string
    public int Display(string txt)
    {
        return int.Parse(txt);
    }

    // string Test(int x) - Returns T=string, Takes T1=int
    public string Test(int x)
    {
        return x.ToString();
    }
}

// Example 1: int from string
MyDel3<int, string> d1 = op.Display;
int result = d1("123");
Console.WriteLine(result);  // Output: 123

// Example 2: string from int
MyDel3<string, int> d2 = op.Test;
string text = d2(456);
Console.WriteLine(text);  // Output: "456"
```

Visual Representation

```
MyDel3<int, string> d1 = op.Display;
        ↑      ↑
        │      └── T1 = string (parameter type)
        └───────── T = int (return type)

d1("123") → op.Display("123") → returns int(123)
```

```
MyDel3<string, int> d2 = op.Test;
         ↑       ↑
         │       └── T1 = int (parameter type)
         └────────── T = string (return type)

d2(456) → op.Test(456) → returns "456"
```

Why Generic Delegates?

```
// Without generics - need separate delegates:
delegate int IntStringDel(string x);
delegate string StringIntDel(int x);
delegate double DoubleFloatDel(float x);
// ... many more!

// With generics - ONE delegate for all!
delegate T MyDel3<T, T1>(T1 x);

// Can be used for any combination!
MyDel3<int, string> d1;
MyDel3<string, int> d2;
MyDel3<double, float> d3;
MyDel3<Student, int> d4;
// ... unlimited combinations!
```

Built-in Delegates

Definition

C# provides three built-in generic delegates that cover most use cases: **Func**, **Action**, and **Predicate**.

1. Func< T > - Methods that Return a Value

```
// Func<TResult>
Func<int> getNumber = () => 42;
int num = getNumber();  // 42

// Func<T, TResult>
```

```
Func<int, int> square = x => x * x;
int result = square(5);  // 25

// Func<T1, T2, TResult>
Func<int, int, int> add = (x, y) => x + y;
int sum = add(3, 5);  // 8

// Func<T1, T2, T3, TResult>
Func<int, int, int, int> calculate = (a, b, c) => (a + b) * c;
int answer = calculate(2, 3, 4);  // 20

// Up to Func<T1, T2, ..., T16, TResult>
```

Func in Your Code

```
Operation op = new Operation();

// Instead of: MyDel d = op.Sum;
Func<int, int, int> d = op.Sum;
//   ↑    ↑    ↑
//   │    │    └── Return type (always last!)
//   │    └──────── Parameter 2 type
//   └───────────── Parameter 1 type

d(5, 4);  // Output: sum=9
```

Your Calc Method with Func

```
// Old version with custom delegate:
delegate int MyDel(int x, int y);
static void Calc(int x, int y, MyDel d)
{
    Console.WriteLine(d(x, y));
}

// New version with Func:
static void Calc(int x, int y, Func<int, int, int> d)
{
    Console.WriteLine(d(x, y));
}

// Usage is the same!
Calc(7, 3, op.Sum);       // Output: sum=10, 10
Calc(7, 3, (x, y) => x * y);  // Output: 21
```

2. Action< T> - Methods that Return void

```
// Action (no parameters, no return)
Action sayHello = () => Console.WriteLine("Hello!");
sayHello();  // Output: Hello!

// Action<T>
Action<string> greet = name => Console.WriteLine($"Hello, {name}!");
greet("Ali");  // Output: Hello, Ali!

// Action<T1, T2>
Action<int, int> display = (x, y) => Console.WriteLine($"{x} + {y} = {x+y}");
display(5, 3);  // Output: 5 + 3 = 8

// Up to Action<T1, T2, ..., T16>
```

Action in Your Code

```
Operation op = new Operation();

// Display method: void Display(int x, int y)
Action<int, int> d1 = op.Display;
//     ↑    ↑
//     │    └── Parameter 2 type
//     │
//     └──────── Parameter 1 type
// NO return type (void)

d1(5, 10);  // Calls op.Display(5, 10)
```

3. Predicate< T> - Methods that Return bool

```
// Predicate<T> always returns bool
// Used for testing conditions

Predicate<int> isEven = x => x % 2 == 0;
Console.WriteLine(isEven(4));  // True
Console.WriteLine(isEven(5));  // False

Predicate<string> isLong = s => s.Length > 5;
Console.WriteLine(isLong("Hello"));      // False
Console.WriteLine(isLong("Hello World"));  // True

Predicate<Student> isAdult = s => s.Age >= 18;
```

```
Student student = new Student(1, "Ali", 20);
Console.WriteLine(isAdult(student));  // True
```

Predicate with Collections

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Find first even number
int firstEven = numbers.Find(x => x % 2 == 0);  // 2

// Find all even numbers
List<int> evens = numbers.FindAll(x => x % 2 == 0);
// evens = [2, 4, 6, 8, 10]

// Check if all are positive
bool allPositive = numbers.TrueForAll(x => x > 0);  // True

// Check if any are greater than 5
bool anyLarge = numbers.Exists(x => x > 5);  // True
```

Comparison Table

Delegate	Signature	Purpose	Example
Func	Returns T	Method with return	`Func<int, int> square = x => x * x;`
Action	Returns void	Method without return	`Action<string> print = s => Console.WriteLine(s);`
Predicate	Returns bool	Condition test	`Predicate<int> isEven = x => x % 2 == 0;`

When to Use Each

✓ **Guidelines**

Use Func< T> when:

- Method returns a value
- Need to process and return result
- Transformation operations

Use Action< T> when:

- Method returns void
- Performing an action (side effect)
- Display, log, save operations

Use Predicate< T> when:

- Testing a condition
- Filtering collections
- Validation logic

Memory Diagrams

Single Delegate in Memory

```
Operation op = new Operation();
MyDel d = op.Sum;
```

```
STACK:
╔══════════════════════════════════════════════════════════════╗
║ Main Method Frame                                            ║
║  ┌─────────────────────────────────────────────────────────┐ ║
║  │ op @ 0x1000 │ → 0xA000                                  │ ║
║  │ d  @ 0x1008 │ → 0xB000                                  │ ║
║  └─────────────────────────────────────────────────────────┘ ║
╚════════════════════════════════│════════════════│════════════╝
                                 │                │
                                 ↓                ↓
HEAP:
╔══════════════════════════════════════════════════════════════╗
║ @ 0xA000: Operation Object                                   ║
║  ┌─────────────────────────────────────────────────────────┐ ║
║  │ Type: Operation                                         │ ║
║  │ Methods: Sum, Sub, Mul, Div, ...                        │ ║
║  └─────────────────────────────────────────────────────────┘ ║
║                                                              ║
║ @ 0xB000: Delegate Object (MyDel)                            ║
║  ┌─────────────────────────────────────────────────────────┐ ║
║  │ Type: MyDel                                             │ ║
║  │ _target: → 0xA000 (Operation instance)                  │ ║
║  │ _methodPtr: → Operation.Sum method                      │ ║
║ │ _invocationList: null (single-cast)                       │ ║
║ └───────────────────────────────────────────────────────────┘ ║
╚═══════════════════════════════════════════════════════════════╝

When d(5, 3) is called:
┌─────────────────────────────────────────┐
│ 1. Get _target (0xA000)                 │
│ 2. Get _methodPtr (Sum method)          │
│ 3. Call _target.Sum(5, 3)               │
│ 4. Return result                        │
└─────────────────────────────────────────┘
```

Multicast Delegate in Memory

```
MyDel d = op.Mul;
d += op.Sub;
d += op.Sub;
```

```
HEAP:
╔═══════════════════════════════════════════════════════════════╗
║ @ 0xB000: Delegate Object (MyDel) - Multicast                 ║
║ ┌───────────────────────────────────────────────────────────┐ ║
║ │ Type: MyDel                                               │ ║
║ │ _target: null (multicast)                                 │ ║
║ │ _methodPtr: null (multicast)                              │ ║
║ │ _invocationList: → 0xC000 ──────────┐                     │ ║
║ └─────────────────────────────────────│─────────────────────┘ ║
║                                       │                       ║
║                                       ↓                       ║
║ @ 0xC000: Delegate[] Array (Invocation List)                  ║
║ ┌───────────────────────────────────────────────────────────┐ ║
║ │ [0] → 0xC100 (Delegate to op.Mul)                         │ ║
║ │ [1] → 0xC200 (Delegate to op.Sum)                         │ ║
║ │ [2] → 0xC300 (Delegate to op.Sub)                         │ ║
║ └───────────────────────────────────────────────────────────┘ ║
║         │            │            │                           ║
║         ↓            ↓            ↓                           ║
║ @ 0xC100: Delegate                                            ║
║ ┌───────────────────────────────────────────────────────────┐ ║
║ │ _target: → 0xA000 (op)                                    │ ║
║ │ _methodPtr: → Operation.Mul                               │ ║
║ └───────────────────────────────────────────────────────────┘ ║
║                                                               ║
║ @ 0xC200: Delegate                                            ║
║  ┌─────────────────────────────────────────────────────────┐ ║
║  │ _target: → 0xA000 (op)                                  │ ║
║  │ _methodPtr: → Operation.Sum                             │ ║
║  └─────────────────────────────────────────────────────────┘ ║
║                                                              ║
║ @ 0xC300: Delegate                                           ║
║  ┌─────────────────────────────────────────────────────────┐ ║
║  │ _target: → 0xA000 (op)                                  │ ║
║  │ _methodPtr: → Operation.Sub                             │ ║
║  └─────────────────────────────────────────────────────────┘ ║
╚══════════════════════════════════════════════════════════════╝

When d(7, 3) is called:
┌─────────────────────────────────────────┐
│ 1. Loop through _invocationList         │
│ 2. Call [0]: op.Mul(7, 3) → 21          │
│ 3. Call [1]: op.Sum(7, 3) → 10          │
│ 4. Call [2]: op.Sub(7, 3) → 4           │
│ 5. Return last result (4)               │
└─────────────────────────────────────────┘
```

Lambda/Anonymous Method Memory

```
int factor = 10;
MyDel d = (x, y) => (x + y) * factor;
```

```
HEAP:
╔══════════════════════════════════════════════════════════════╗
║ @ 0xB000: Delegate Object                                    ║
║  ┌─────────────────────────────────────────────────────────┐ ║
║  │ Type: MyDel                                             │ ║
║  │ _target: → 0xD000 (Closure object) ────────┐            │ ║
║  │ _methodPtr: → Generated lambda method      │            │ ║
║  └────────────────────────────────────────────│────────────┘ ║
║                                               │              ║
║                                               ↓              ║
║ @ 0xD000: Closure Object (Generated Class)                   ║
║  ┌─────────────────────────────────────────────────────────┐ ║
║  │ Captured Variables:                                     │ ║
║  │  factor: 10  ← Captured from outer scope!               │ ║
║  │                                                         │ ║
║  │ Generated Method:                                       │ ║
║  │  int Lambda(int x, int y)                               │ ║
║  │  {                                                      │ ║
║  │      return (x + y) * this.factor;                      │║
║  │  }                                                      │║
║  └─────────────────────────────────────────────────────────┘║
╚═════════════════════════════════════════════════════════════╝

Closure (Captured Variables):
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━


Lambda captures 'factor' from outer scope.
Compiler generates a class to hold captured variables.
Lambda becomes a method in that class.

Example:
int factor = 10;
MyDel d = (x, y) => (x + y) * factor;
int result = d(5, 3);  // (5 + 3) * 10 = 80
```

Real-World Examples

Example 1: Calculator with Strategy Pattern

```csharp
class Calculator
{
    // Use delegate as strategy
    public int Execute(int a, int b, Func<int, int, int> operation)
    {
        return operation(a, b);
    }
}

// Usage
Calculator calc = new Calculator();

int sum = calc.Execute(10, 5, (x, y) => x + y);       // 15
int diff = calc.Execute(10, 5, (x, y) => x - y);      // 5
int product = calc.Execute(10, 5, (x, y) => x * y);   // 50
int quotient = calc.Execute(10, 5, (x, y) => x / y); // 2
int power = calc.Execute(2, 3, (x, y) => (int)Math.Pow(x, y));  // 8
```

Example 2: Custom Sorting

```
List<Student> students = new List<Student>
{
    new Student(3, "Ali", 22),
    new Student(1, "Sara", 20),
    new Student(2, "Omar", 25)
};

// Sort by Id
students.Sort((s1, s2) => s1.Id.CompareTo(s2.Id));
// Result: Sara(1), Omar(2), Ali(3)

// Sort by Name
students.Sort((s1, s2) => s1.Name.CompareTo(s2.Name));
// Result: Ali, Omar, Sara

// Sort by Age (descending)
students.Sort((s1, s2) => s2.Age.CompareTo(s1.Age));
// Result: Omar(25), Ali(22), Sara(20)
```

Example 3: Event Handling Simulation

```
// Simulate button click handler
Action<string> OnButtonClick = null;

// Subscribe handlers
OnButtonClick += message => Console.WriteLine($"Handler 1: {message}");
OnButtonClick += message => Console.WriteLine($"Handler 2: {message}");
OnButtonClick += message => Console.WriteLine($"Handler 3: {message}");

// Trigger event
OnButtonClick?.Invoke("Button was clicked!");

// Output:
// Handler 1: Button was clicked!
// Handler 2: Button was clicked!
// Handler 3: Button was clicked!
```

Example 4: LINQ-style Operations

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Where (filter)
List<int> evens = numbers.Where(n => n % 2 == 0).ToList();
// [2, 4, 6, 8, 10]
// Select (map/transform)
List<int> squared = numbers.Select(n => n * n).ToList();
// [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

// Aggregate (reduce)
int sum = numbers.Aggregate((total, n) => total + n);
// 55

// Chain operations
var result = numbers
    .Where(n => n % 2 == 0)        // Get evens: [2,4,6,8,10]
    .Select(n => n * n)             // Square them: [4,16,36,64,100]
    .Where(n => n > 20)             // Filter > 20: [36,64,100]
    .Sum();                         // Sum: 200
```

Example 5: Retry Logic with Delegates

```
void RetryOperation(Action operation, int maxRetries)
{
    int attempt = 0;
    while (attempt < maxRetries)
    {
        try
        {
            operation();
            Console.WriteLine("Operation succeeded!");
            return;
        }
        catch (Exception ex)
        {
            attempt++;
            Console.WriteLine($"Attempt {attempt} failed: {ex.Message}");
            if (attempt >= maxRetries)
            {
                Console.WriteLine("Max retries reached. Operation failed.");
                throw;
            }
        }
    }
}

// Usage
RetryOperation(() =>
{
    // Simulate operation that might fail
    if (new Random().Next(2) == 0)
        throw new Exception("Random failure");

    Console.WriteLine("Doing important work...");
}, maxRetries: 3);
```

Summary & Best Practices

Key Concepts

✓ **Delegate Fundamentals**

What are delegates?

- Type-safe function pointers
- Can point to methods with matching signature
- Can be passed as parameters
- Support multicast (multiple methods)

Why use delegates?

- ✅ Callback mechanisms
- ✅ Event handling
- ✅ Flexible design patterns
- ✅ LINQ and functional programming
- ✅ Strategy pattern implementation

Syntax Progression

```
// 1. Custom delegate (old way)
delegate int MyDel(int x, int y);
MyDel d = op.Sum;

// 2. Built-in delegate (better)
Func<int, int, int> d = op.Sum;

// 3. Anonymous method
Func<int, int, int> d = delegate(int x, int y) { return x + y; };
```

```
// 4. Lambda expression (best!)
Func<int, int, int> d = (x, y) => x + y;
```

When to Use What

Scenario	Use
Need custom delegate	Define your own
Method returns value	`Func<T>`
Method returns void	`Action<T>`
Method returns bool	`Predicate<T>`
Simple inline logic	Lambda `(x) => ...`
One-time use method	Anonymous method

Common Patterns

```
// 1. Callback pattern
void ProcessData(int[] data, Action<int> callback)
{
    foreach (int item in data)
        callback(item);
}

// 2. Strategy pattern
int Calculate(int a, int b, Func<int, int, int> strategy)
{
    return strategy(a, b);
}

// 3. Filter pattern
List<T> Filter<T>(List<T> items, Predicate<T> condition)
{
    return items.FindAll(condition);
}

// 4. Transform pattern
List<TResult> Transform<T, TResult>(List<T> items, Func<T, TResult> transform)
{
    return items.Select(transform).ToList();
}
```

End of Documentation

Key Takeaways

- **Delegates** are type-safe function pointers
- **Multicast** delegates can call multiple methods
- **Lambda expressions** provide concise syntax
- **Func, Action, Predicate** cover 99% of use cases
- **Closures** capture variables from outer scope
- Essential for **events**, **LINQ**, and **callbacks**

Abdullah Ali

Contact : +201012613453