Lecture 06 - Practice Tasks

Delegates, Lambda Expressions & Events

Created by: ITI

Tasks Overview

#	Task Name	Topic
1	Calculator Delegate	Basic Delegate Usage
2	Multicast Delegate	+= and -= Operators
3	Array Filter Delegate	Delegate as Parameter
4	Anonymous Method	Inline Delegate (C# 2.0)
5	Lambda Expression Filter	Lambda Syntax (C# 3.0)
6	Lambda Sort	Custom Sorting
7	Temperature Monitor Events	Event Sender/Listener
8	Button Click Events	Complete Event System

Task 1: Calculator Delegate

Description

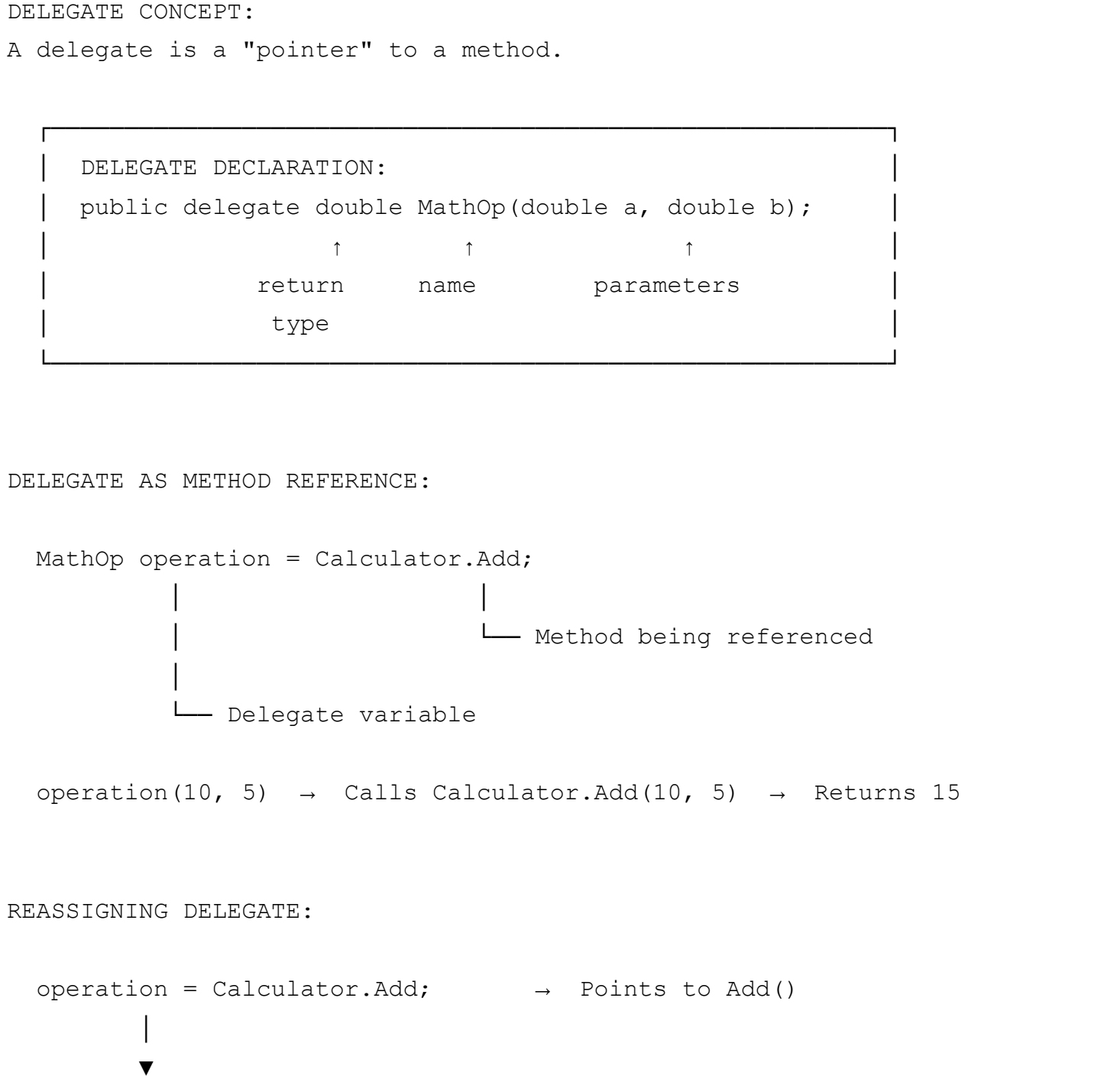
Create a delegate that can reference methods with the signature `double Method(double, double)` . Create Add, Subtract, Multiply, and Divide methods and use the delegate to call them.

A delegate is like a variable that holds a reference to a method.

Example

```
// Declare delegate type
public delegate double MathOperation(double a, double b);

// Create delegate instance
MathOperation operation = Calculator.Add;
double result = operation(10, 5);  // result = 15

// Reassign to different method
operation = Calculator.Multiply;
result = operation(10, 5);  // result = 50
```

Illustration



```
  operation = Calculator.Multiply;  →  Now points to Multiply()
        |
        ▼
  operation = Calculator.Divide;    →  Now points to Divide()


SAME DELEGATE, DIFFERENT METHODS!
```

Task 2: Multicast Delegate

Description

Create a delegate that holds references to MULTIPLE methods. Use the `+=` operator to add methods and `-=` to remove them. When invoked, all methods are called in order.

Example

```
NotifyHandler notify = SendEmail;
notify += SendSMS;
notify += LogToFile;

notify("Order confirmed!");
// Output:
// Email sent: Order confirmed!
// SMS sent: Order confirmed!
// Logged: Order confirmed!

notify -= SendSMS;
notify("Shipped!");
// Now only Email and Log are called
```

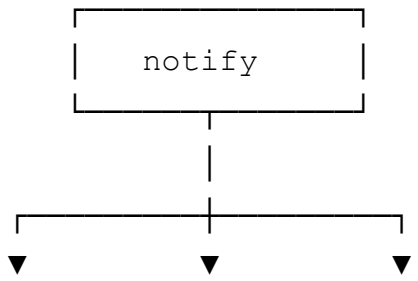
Illustration

```
MULTICAST DELEGATE:
One delegate → Multiple methods


ADDING METHODS (+=):
```

```
  notify = SendEmail;          notify ──► SendEmail
  notify += SendSMS;           notify ──┬► SendEmail
                                        └► SendSMS
  notify += LogToFile;         notify ──┬► SendEmail
                                        ├► SendSMS
                                        └► LogToFile
```

INVOKING MULTICAST DELEGATE:

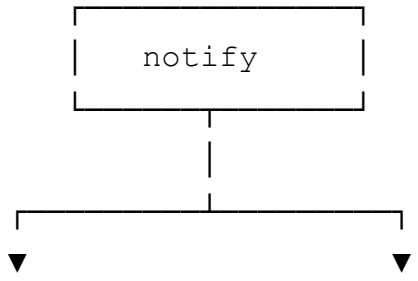
```
  notify("Hello!");
```




```
  SendEmail   SendSMS   LogToFile
  ("Hello")  ("Hello")  ("Hello")

  All 3 methods called in order!
```

REMOVING METHODS (-=):

```
  notify -= SendSMS;
```




```
  SendEmail           LogToFile

  SMS removed, only 2 methods remain.
```

Task 3: Array Filter Delegate

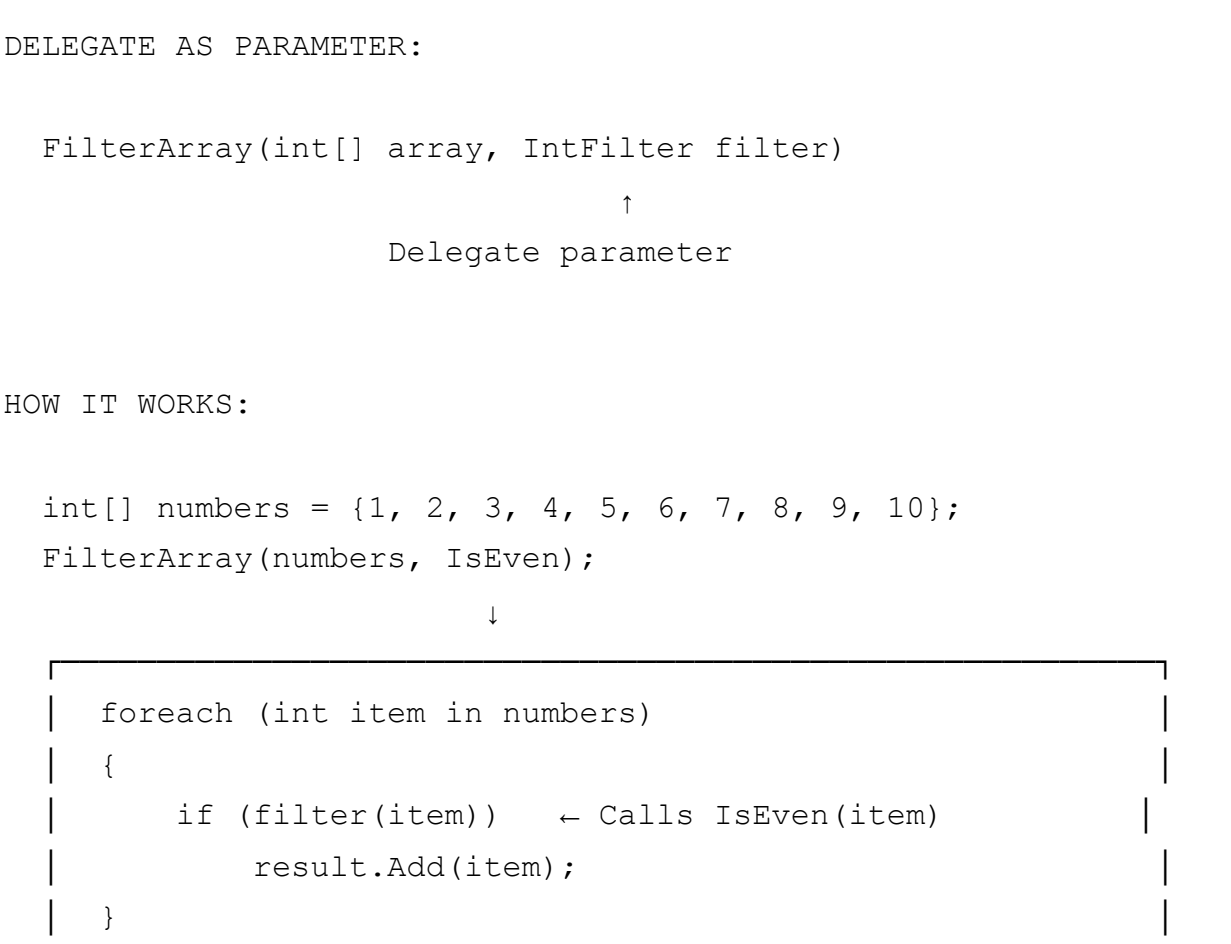
Description

Create a `FilterArray` method that accepts a delegate as a parameter. The delegate determines which elements to keep. This pattern separates the filtering algorithm from the filter criteria.

Example

```
public delegate bool IntFilter(int value);

public static int[] FilterArray(int[] array, IntFilter filter)
{
    // For each item, call filter(item)
    // If true, keep the item
}

int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int[] evens = FilterArray(numbers, IsEven);      // [2,4,6,8,10]
int[] odds = FilterArray(numbers, IsOdd);        // [1,3,5,7,9]
int[] big = FilterArray(numbers, IsGreaterThan5);// [6,7,8,9,10]
```

Illustration



```
  └──────────────────────────────────────────────────────┘

  Item: 1  →  IsEven(1) = false  →  Skip
  Item: 2  →  IsEven(2) = true   →  Add to result
  Item: 3  →  IsEven(3) = false  →  Skip
  Item: 4  →  IsEven(4) = true   →  Add to result
  ...

  Result: [2, 4, 6, 8, 10]


DIFFERENT FILTERS, SAME METHOD:

  FilterArray(nums, IsEven)         →  [2,4,6,8,10]
  FilterArray(nums, IsOdd)          →  [1,3,5,7,9]
  FilterArray(nums, IsGreaterThan5) →  [6,7,8,9,10]
  FilterArray(nums, IsPrime)        →  [2,3,5,7]

  Algorithm stays same, only filter changes!
```

Task 4: Anonymous Method

Description

Use anonymous methods (C# 2.0) to create delegate implementations inline, without declaring a separate named method. Use the `delegate` keyword followed by parameters and body.

Example

```
// Named method approach:
bool IsEven(int n) { return n % 2 == 0; }
NumberFilter filter = IsEven;

// Anonymous method approach:
NumberFilter filter = delegate(int n)
{
    return n % 2 == 0;
};
```

```
// Use directly as argument:
int[] evens = FilterArray(nums, delegate(int n) { return n % 2 == 0; });
```

Illustration

ANONYMOUS METHOD SYNTAX:

```
┌──────────────────────────────────────────────────────────┐
│  delegate(parameters)                                    │
│  {                                                       │
│      // method body                                      │
│      return value;                                       │
│  }                                                       │
└──────────────────────────────────────────────────────────┘
```

COMPARISON:

```
  NAMED METHOD:                          ANONYMOUS METHOD:
┌──────────────────────────┐          ┌──────────────────────────┐
│ // Separate method       │          │ // Inline with delegate  │
│ bool IsEven(int n)       │          │                          │
│ {                        │          │ delegate(int n)          │
│   return n % 2 == 0;     │    →     │ {                        │
│ }                        │          │   return n % 2 == 0;     │
│                          │          │ }                        │
│ filter = IsEven;         │          │                          │
└──────────────────────────┘          └──────────────────────────┘
```

WHEN TO USE:

- One-time use (don't need method elsewhere)
- Keep code where it's used
- Short logic

CLOSURE (Accessing Outer Variables):

```
  int threshold = 5;
```

```
NumberFilter f = delegate(int n)
{
    return n > threshold;  ← Can access outer variable!
};

f(10);  // true (10 > 5)
```

Task 5: Lambda Expression Filter

Description

Use lambda expressions (C# 3.0) as an even shorter syntax for anonymous methods. Use the `=>` (arrow) operator. Works great with List methods like Find, FindAll, Exists.

Example

```
// Anonymous method:
NumberFilter f = delegate(int n) { return n % 2 == 0; };

// Lambda expression (much shorter!):
NumberFilter f = n => n % 2 == 0;

// With List<T> methods:
List<int> numbers = new List<int> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int first = numbers.Find(n => n > 5);           // 6
List<int> evens = numbers.FindAll(n => n % 2 == 0);  // [2,4,6,8,10]
bool hasNeg = numbers.Exists(n => n < 0);       // false
```

Illustration

```
LAMBDA SYNTAX:

  parameters => expression

  ┌──────────────────────────────────────────────────────────┐
  │  n => n % 2 == 0                                         │
  │  ↑    ↑                                                  │
  │  │    └── Expression (body)                              │
```

```
  │  └── Parameter                                                  │
  │                                                                 │
  │  Read as: "n goes to n % 2 == 0"                                │
  └─────────────────────────────────────────────────────────────────┘


EVOLUTION OF SYNTAX:

  1. Named Method:
     bool IsEven(int n) { return n % 2 == 0; }

  2. Anonymous Method (C# 2.0):
     delegate(int n) { return n % 2 == 0; }

  3. Lambda Expression (C# 3.0):
     n => n % 2 == 0

     ↓ Shorter and cleaner!


LAMBDA VARIATIONS:

  Single parameter:     n => n > 0
  Multiple parameters:  (a, b) => a + b
  No parameters:        () => DateTime.Now
  Statement block:      n => { if (n > 0) return true; return false; }


WITH LIST<T>:

  numbers.Find(n => n > 10)          // First match
  numbers.FindAll(n => n % 2 == 0)   // All matches
  numbers.Exists(n => n < 0)         // Any match?
  numbers.RemoveAll(n => n > 100)    // Remove matches
  numbers.ForEach(n => Console.WriteLine(n))  // Action on each
```

Task 6: Lambda Sort

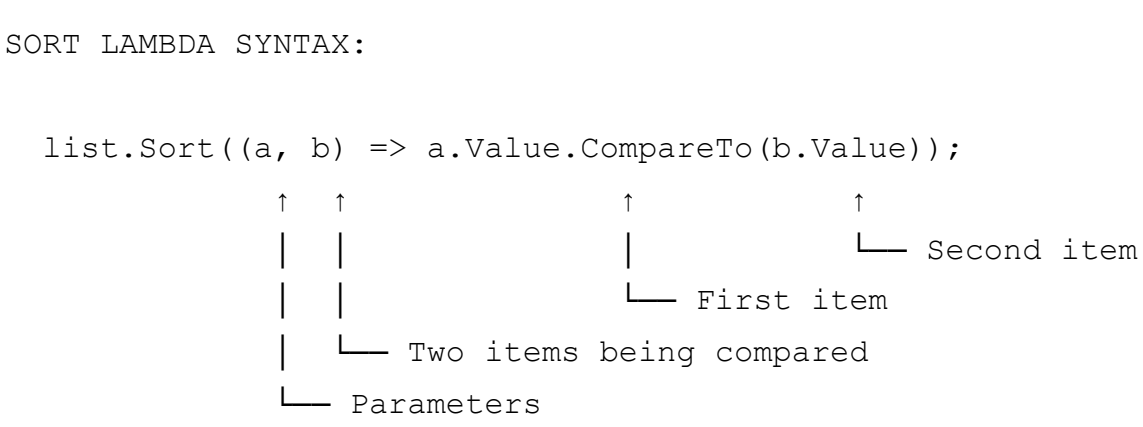
Description

Use lambda expressions with the `Sort` method to sort lists by different criteria. The lambda compares two items and returns negative (a before b), zero (equal), or positive (a after b).

Example

```
List<Person> people = ...;

// Sort by Age (ascending)
people.Sort((a, b) => a.Age.CompareTo(b.Age));

// Sort by Age (descending)
people.Sort((a, b) => b.Age.CompareTo(a.Age));  // Note: b, a

// Sort by Name
people.Sort((a, b) => a.Name.CompareTo(b.Name));

// Sort by multiple criteria
people.Sort((a, b) => {
    int result = a.Department.CompareTo(b.Department);
    if (result != 0) return result;
    return a.Name.CompareTo(b.Name);
});
```

Illustration



```
  |  > 0   →  a should come AFTER b                                  |
  └──────────────────────────────────────────────────────────────────┘


ASCENDING vs DESCENDING:

  ASCENDING (low to high):
  (a, b) => a.Age.CompareTo(b.Age)
            ↑                ↑
          first           second

  DESCENDING (high to low):
  (a, b) => b.Age.CompareTo(a.Age)
            ↑                ↑
         second           first
         (swapped!)


EXAMPLE:

  People: Ahmed(30), Sara(25), Omar(35)

  Sort ascending by age:
  (a, b) => a.Age.CompareTo(b.Age)

  Result: Sara(25), Ahmed(30), Omar(35)
```

Task 7: Temperature Monitor Events

Description

Create an event-based system with a temperature sensor (sender) that fires events when temperature changes, and monitor classes (listeners) that respond to those events.

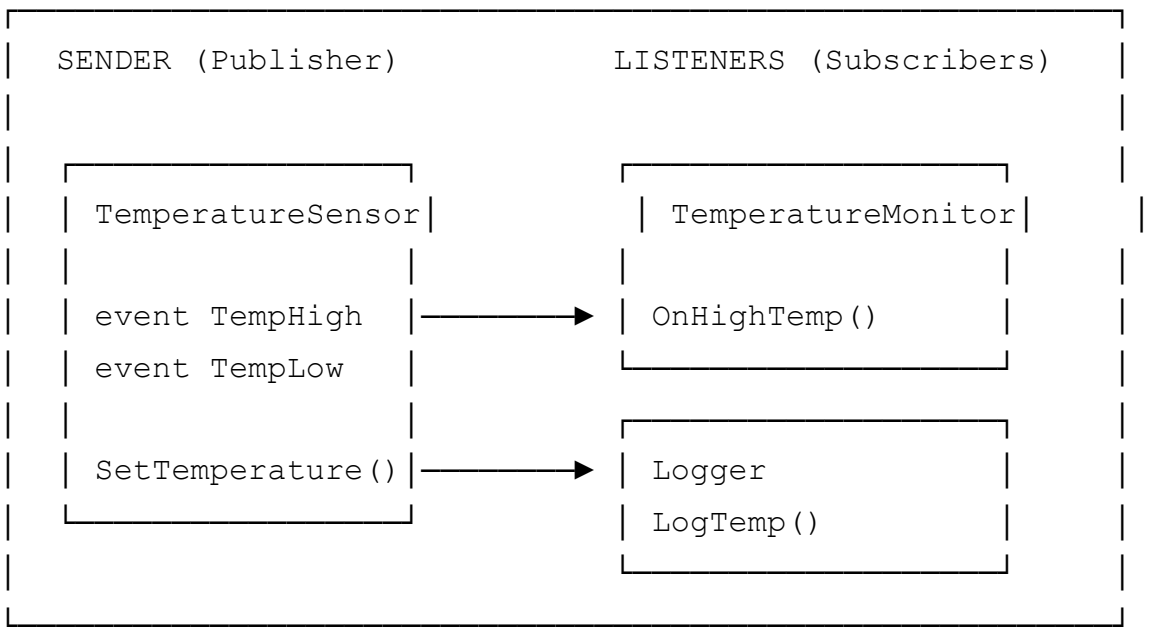
Example

```
// SENDER class:
public event TemperatureHandler TemperatureHigh;
```

```
public void SetTemperature(double temp)
{
    if (temp > 30)
    {
        if (TemperatureHigh != null)
            TemperatureHigh("Warning!", temp);
    }
}

// LISTENER subscribes:
sensor.TemperatureHigh += monitor.OnHighTemperature;

// Handler method:
public void OnHighTemperature(string msg, double temp)
{
    Console.WriteLine($"Alert: {temp}°C - {msg}");
}
```

Illustration

EVENT PATTERN:



EVENT FLOW:

1. Listener subscribes:
   sensor.TempHigh += monitor.OnHighTemp;

```
  2. Something happens:
     sensor.SetTemperature(35);

  3. Sender fires event:
     if (TempHigh != null)
         TempHigh("Warning!", 35);

  4. Listener responds:
     OnHighTemp("Warning!", 35)
     → "Alert: High temperature!"


SUBSCRIBE / UNSUBSCRIBE:

  sensor.TempHigh += handler;   // Subscribe (+=)
  sensor.TempHigh -= handler;   // Unsubscribe (-=)
```

Task 8: Button Click Events

Description

Create a complete event system simulating GUI buttons. Each button has a Click event, and multiple handlers can subscribe. Use both methods and lambda expressions as event handlers.

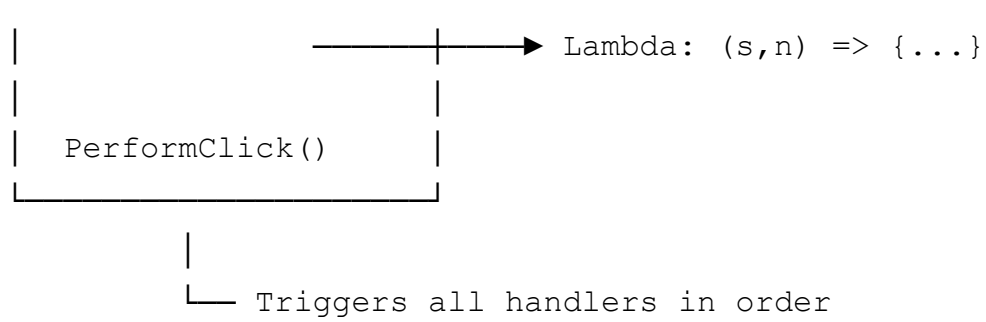
Example

```
// Button class with event
public event ClickHandler Click;

public void PerformClick()
{
    if (Click != null)
        Click(this, buttonName);
}

// Subscribe handlers
button.Click += handler.OnClick;
button.Click += logger.LogClick;
```

```
button.Click += (sender, name) => Console.WriteLine($"Clicked: {name}");

// Trigger
button.PerformClick();  // All 3 handlers called!
```

Illustration

```
COMPLETE EVENT SYSTEM:

  1. DECLARE DELEGATE:

  ┌──────────────────────────────────────────────────────────┐
  │ public delegate void ClickHandler(object sender,         │
  │                                   string buttonName);    │
  └──────────────────────────────────────────────────────────┘

  2. DECLARE EVENT (in Button class):

  ┌──────────────────────────────────────────────────────────┐
  │ public event ClickHandler Click;                         │
  └──────────────────────────────────────────────────────────┘

  3. FIRE EVENT:

  ┌──────────────────────────────────────────────────────────┐
  │ if (Click != null)                                       │
  │     Click(this, name);                                   │
  └──────────────────────────────────────────────────────────┘

  4. SUBSCRIBE:

  ┌──────────────────────────────────────────────────────────┐
  │ button.Click += handler.OnClick;                         │
  │ button.Click += logger.LogClick;                         │
  │ button.Click += (s, n) => Console.WriteLine(n);          │
  └──────────────────────────────────────────────────────────┘


BUTTON WITH MULTIPLE HANDLERS:

        ┌──────────────────┐
        │      Button      │
        │                  │
        │  Click event ────┼──► FormHandler.OnClick()
        │              ────┼──► Logger.LogClick()
```



EVENT vs DELEGATE:

- Event is a RESTRICTED delegate
- Only sender class can fire (invoke) the event
- Subscribers can only += or -= (cannot invoke)

Good Luck!

Notes:

- Test your code with different scenarios
- Make sure your code compiles without errors
- Understand the evolution: Named → Anonymous → Lambda
- Events follow the publisher-subscriber pattern

Created by: ITI