Complete Guide to C# Events

Table of Contents

What are Events?

> **Definition**
>
> An **event** is a message sent by an object to signal the occurrence of an action.
> Events are built on top of delegates and provide a publish-subscribe pattern.

Real-World Analogy

```
Button Click Event:
═══════════════════════════════════════════

Publisher (Button):
"I was clicked! Notifying all subscribers..."

Subscribers:
1. Logger: "I'll log this click"
2. Counter: "I'll increment my count"
3. Analytics: "I'll track this event"

Button doesn't know/care who's listening.
Subscribers don't know about each other.
```

Events vs Regular Methods

```
WITHOUT Events (Tight Coupling):
┌──────────────────────────────────────────┐
│ Exam class must know about:              │
│  - Every Student                         │
│  - Every Admin                           │
│  - Call each one explicitly              │
│                                          │
│ exam.OnStartExam() {                     │
│     student1.AnswerExam();               │
│     student2.AnswerExam();               │
│     admin1.Monitor();                    │
│     admin2.Monitor();                    │
│ }                                        │
│                                          │
│ ❌ Tightly coupled                       │
│ ❌ Hard to extend                        │
│ ❌ Exam knows too much                   │
└──────────────────────────────────────────┘

WITH Events (Loose Coupling):
┌──────────────────────────────────────────┐
│ Exam class just raises event:            │
│                                          │
│ exam.OnStartExam() {                     │
│     startexam?.Invoke(...);              │
│ }                                        │
│                                          │
│ Students/Admins subscribe:               │
│ exam.startexam += student.AnswerExam;    │
│ exam.startexam += admin.Monitor;         │
│                                          │
│ ✅ Loosely coupled                       │
│ ✅ Easy to extend                        │
│ ✅ Exam doesn't know subscribers         │
└──────────────────────────────────────────┘
```

Events vs Delegates

Key Differences

```
// DELEGATE - No protection
public delegate void MyDelegate();
public MyDelegate myDelegate;  // Public field

// Anyone can:
myDelegate = SomeMethod;       // ❌ Replace all subscribers!
myDelegate.Invoke();           // ❌ Invoke from outside!
myDelegate = null;             // ❌ Clear all subscribers!


// EVENT - Protected
public event MyDelegate myEvent;  // Event

// Only owner can:
myEvent?.Invoke();             // ✅ Only owner can invoke

// Subscribers can only:
myEvent += SomeMethod;         // ✅ Subscribe
myEvent -= SomeMethod;         // ✅ Unsubscribe
myEvent = SomeMethod;          // ❌ ERROR! Cannot replace
```

Comparison Table

Feature	Delegate	Event
Invoke	Anyone	Only owner
Assignment	Anyone	Compile error
+= , -=	Yes	Yes
= (replace)	Yes	No
Protection	None	Encapsulated
Use Case	Callbacks	Publisher-Subscriber

Visual Representation

```
DELEGATE (Public Field):
┌──────────────────────────────────────┐
│ Outside Class                        │
│                                      │
│ exam.startexam = null;  ✅ Allowed!  │
│ exam.startexam();       ✅ Allowed!  │
│                                      │
│ ⚠️Anyone can mess with delegate!      │
```

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

EVENT (Protected):
┌──────────────────────────────────────┐
│ Outside Class                        │
│                                      │
│ exam.startexam = null;   ❌ ERROR!   │
│ exam.startexam();        ❌ ERROR!   │
│ exam.startexam += ...;   ✅ OK!      │
│ exam.startexam -= ...;   ✅ OK!      │
│                                      │
│ ✅ Only subscribe/unsubscribe allowed │
└──────────────────────────────────────┘
```

Event Pattern Components

Standard Event Pattern

```
// 1. EventArgs - Data about event
class ExamArgs : EventArgs
{
    public string Location { get; set; }
    public DateTime StartTime { get; set; }
}

// 2. Publisher - Raises events
class Exam
{
    // Event declaration
    public event EventHandler<ExamArgs> StartExam;

    // Protected method to raise event
    protected virtual void OnStartExam(ExamArgs e)
    {
        StartExam?.Invoke(this, e);
    }

    // Public method that triggers event
    public void BeginExam()
    {
        OnStartExam(new ExamArgs
        {
            Location = "Room 101",
```

```
            StartTime = DateTime.Now.AddMinutes(30);
        });
    }
}

// 3. Subscribers - Handle events
class Student
{
    public void HandleExamStart(object sender, ExamArgs e)
    {
        Exam exam = sender as Exam;
        Console.WriteLine($"Student ready at {e.Location}");
    }
}
```

Pattern Structure

```
Event Pattern Components:
═══════════════════════════════════════

┌─────────────────────────────────────┐
│ 1. EventArgs (Data Container)       │
│    - Holds information about event  │
│    - Inherits from EventArgs        │
│    - Example: location, time, etc.  │
└─────────────────────────────────────┘
                ↓
┌─────────────────────────────────────┐
│ 2. Publisher (Event Raiser)         │
│    - Declares event                 │
│    - Raises event when action occurs│
│    - Example: Exam class            │
└─────────────────────────────────────┘
                ↓
┌─────────────────────────────────────┐
│ 3. Subscribers (Event Handlers)     │
│    - Subscribe to event (+=)        │
│    - Execute when event raised      │
│    - Example: Student, Admin classes│
└─────────────────────────────────────┘
```

EventArgs Class

Purpose

> **Why EventArgs?**
>
> EventArgs is the base class for all event data. It provides a standard way to pass event-specific information.

Your Code Example

```
class examArgs : EventArgs  // ← Should inherit EventArgs
{
    public string location { get; set; }
    public DateTime starttime { get; set; }

    public examArgs(string location, DateTime starttime)
    {
        this.location = location;
        this.starttime = starttime;
    }
}
```

Standard Pattern

```
// Better naming (PascalCase)
class ExamEventArgs : EventArgs
{
    public string Location { get; set; }
    public DateTime StartTime { get; set; }
    public int Duration { get; set; }
    public string SubjectName { get; set; }

    public ExamEventArgs(string location, DateTime startTime)
    {
        Location = location;
        StartTime = startTime;
    }
}
```

Built-in EventArgs

```
// System.EventArgs - Base class (no data)
public class EventArgs
```

```
{
    public static readonly EventArgs Empty = new EventArgs();
}

// Use EventArgs.Empty when no data needed
button.Click?.Invoke(this, EventArgs.Empty);
```

EventHandler<T>

What is EventHandler<T>?

```
// EventHandler<T> is a built-in generic delegate:
public delegate void EventHandler<TEventArgs>(
    object sender,
    TEventArgs e
) where TEventArgs : EventArgs;

// Equivalent to your custom delegate:
delegate void mydel(exam ex, examArgs e);
```

Why Use EventHandler< T > ?

```
// OLD WAY - Custom delegate
delegate void mydel(exam ex, examArgs e);
public event mydel startexam;

// NEW WAY - Built-in EventHandler <T\>
public event EventHandler<examArgs> startexam;

// Benefits:
// ✅ Standard pattern recognized by everyone
// ✅ No need to declare custom delegate
// ✅ Works with all .NET event conventions
// ✅ Better tooling support
```

Signature Breakdown

```
EventHandler<ExamArgs> handler = (sender, e) => { };
//              ↑          ↑        ↑      ↑
//              |          |        |      └──── Event data
//              |          |        └─────────── Who raised event
//
```

```
//                 |                    └──────────────────── Lambda parameters
//                 └──────────────────────────────────────── Event data type

// sender: object - The object that raised the event
// e: ExamArgs    - Data about the event
```

Evolution of Your Code

```
// Version 1: Custom delegate
delegate void mydel(exam ex, examArgs e);
public event mydel startexam;

// Version 2: Action (non-standard)
public event Action<exam, examArgs> startexam;

// Version 3: EventHandler<T> (BEST! ✨)
public event EventHandler<examArgs> startexam;
```

Publisher-Subscriber Pattern

Complete Example from Your Code

```
// PUBLISHER: Exam class
class exam
{
    // 1. Declare event
    public event EventHandler<examArgs> startexam;

    // 2. Method to raise event
    public void OnStartExam()
    {
        // Null-conditional operator: only invoke if subscribers exist
        startexam?.Invoke(this, new examArgs("lec3",
DateTime.Now.AddMinutes(15)));
        //         ↑       ↑     ↑
        //         |       |     └── Event data
        //         |       └──────── Sender (this exam instance)
        //         └──────────────── Safe invoke
    }
}

// SUBSCRIBER 1: Student class
```

```
class Student
{
    public void answerexam(object sender, examArgs e)
    {
        exam ex = sender as exam;  // Get exam that raised event
        Console.WriteLine(ex);
        Console.WriteLine($"student {name} start answer exam @{e.location}");
    }
}

// SUBSCRIBER 2: Admin class
class Admin
{
    public void monitior(object sender, examArgs e)
    {
        exam ex = sender as exam;  // Get exam that raised event
        Console.WriteLine($"admin {name} start monitor {ex.subjectname} exam
location {e.location} @ {e.starttime.ToShortTimeString()}");
    }
}

// MAIN: Wire it all together
exam ex = new exam(1, "C#", 90, questions);

Student st = new Student(2, "ali", 20);
ex.startexam += st.answerexam;  // Subscribe

Admin ad = new Admin(3, "mahmoud", 40);
ex.startexam += ad.monitior;     // Subscribe

ex.OnStartExam();  // Raises event → notifies ALL subscribers
```

Execution Flow

```
Step-by-Step Execution:
═══════════════════════════════════════════

1. Create exam object
   ┌──────────────────────────────────┐
   │ exam ex = new exam(...);         │
   │ → Creates Exam instance          │
   │ → startexam event = null (no subs)│
   └──────────────────────────────────┘

2. Student subscribes
```

```
ex.startexam += st.answerexam;
→ Adds student handler to event
→ startexam now has 1 subscriber
```

3. Admin subscribes

```
ex.startexam += ad.monitior;
→ Adds admin handler to event
→ startexam now has 2 subscribers
```

4. Raise event

```
ex.OnStartExam();
→ Invokes startexam event
→ Calls ALL subscribed handlers
```

↓

```
st.answerexam(ex, eventArgs)
→ Student handles event
→ Prints exam details
```

↓

```
ad.monitior(ex, eventArgs)
→ Admin handles event
→ Prints monitoring info
```

Output Explanation

When ex.OnStartExam() is called:

1. Student handler executes:

```
Num:1    Subject:C#     Duration:90 minutes
What is C#?
1-Strong Lang
2-is C-Sharp
3- all above
... (all questions)
```

```
student ali start answer exam @lec3
```

2. Admin handler executes:

```
admin mahmoud start monitor C# exam location lec3
@ 3:45 PM ,exam duration: 90 Mintues ...
```

Memory Diagrams

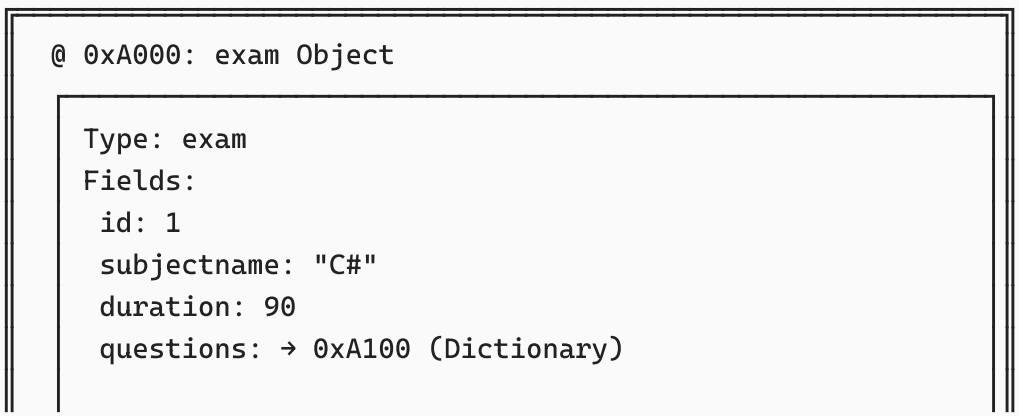
Event Subscription Memory Layout

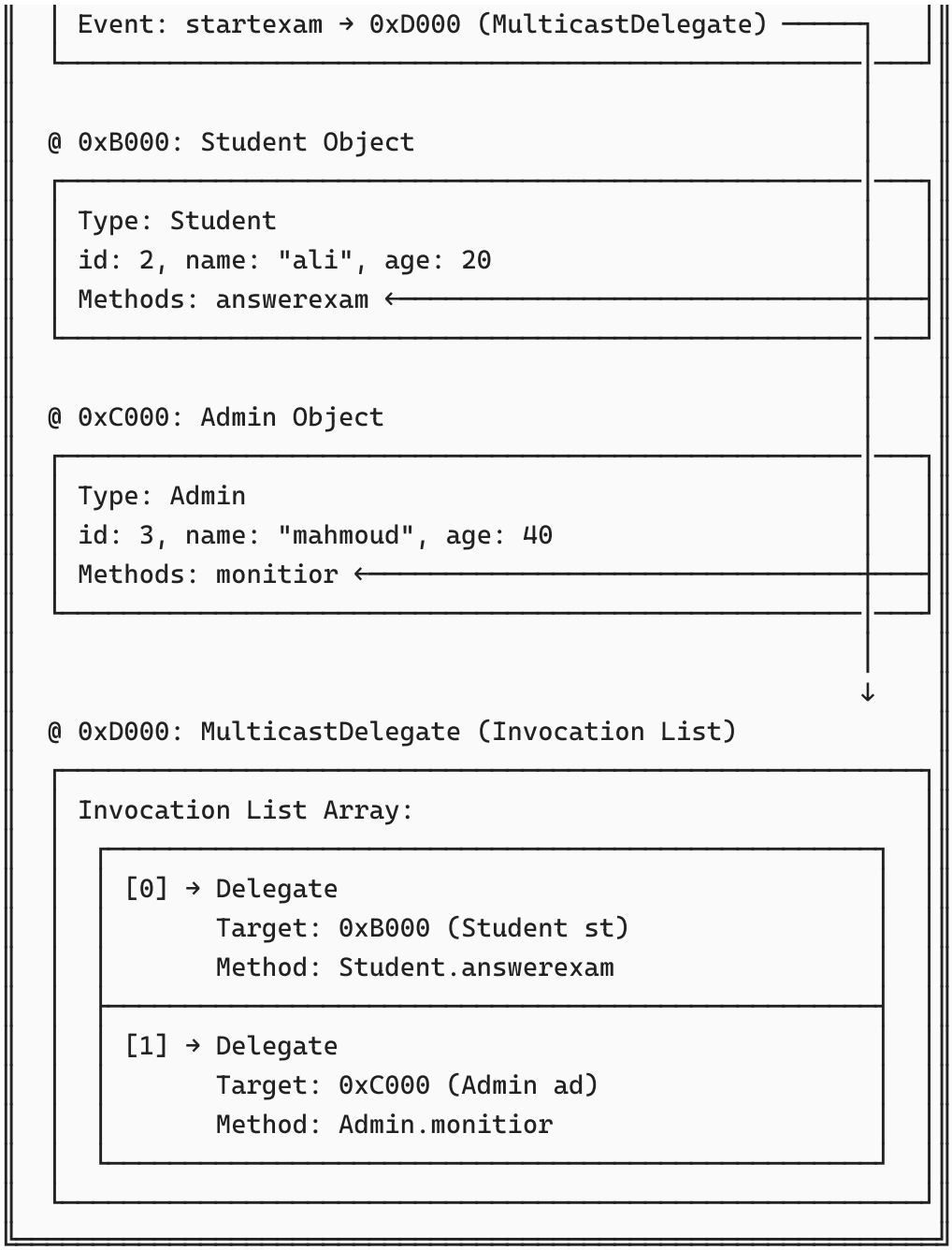
```
exam ex = new exam(1, "C#", 90, questions);
Student st = new Student(2, "ali", 20);
Admin ad = new Admin(3, "mahmoud", 40);

ex.startexam += st.answerexam;
ex.startexam += ad.monitior;
```

STACK:

HEAP:

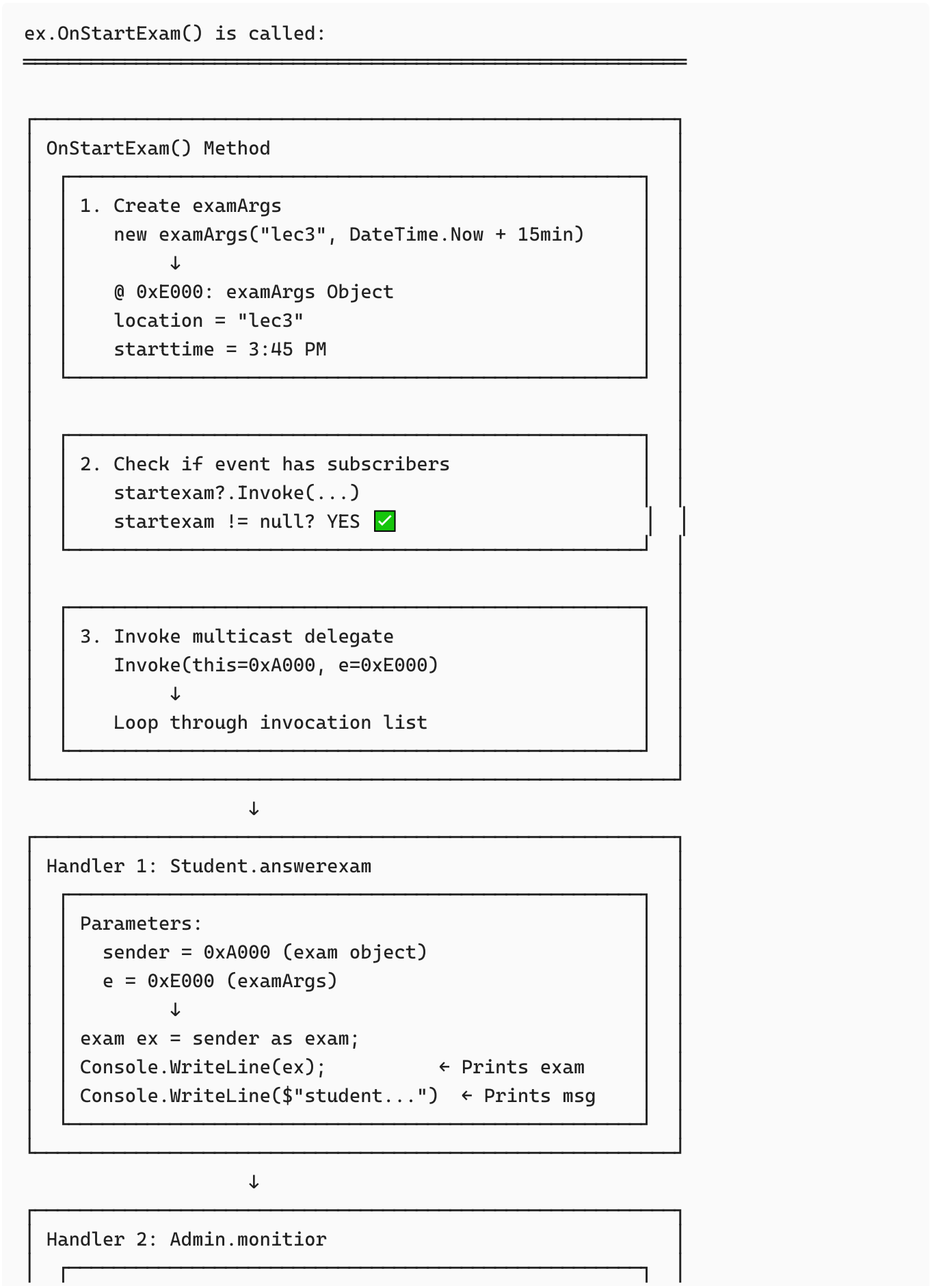




Visual Flow:

```
exam.startexam Event
        │
        ├──→ [0] st.answerexam
        │          ↓
        │     Calls Student.answerexam(exam, examArgs)
        │
        └──→ [1] ad.monitior
                   ↓
              Calls Admin.monitior(exam, examArgs)
```

Event Invocation Flow



```
    Parameters:
      sender = 0xA000 (exam object)
      e = 0xE000 (examArgs - SAME instance!)
            ↓
    exam ex = sender as exam;
    Console.WriteLine($"admin...")  ← Prints msg
```

```
                    ↓
              Event Complete!
```

Complete Code Walkthrough

Step 1: Define EventArgs

```
class examArgs : EventArgs
{
    public string location { get; set; }
    public DateTime starttime { get; set; }

    public examArgs(string location, DateTime starttime)
    {
        this.location = location;
        this.starttime = starttime;
    }
}
```

Purpose: Container for event data (location, time)

Step 2: Publisher Class (exam)

```
class exam
{
    // Properties
    public int id { get; set; }
    public string subjectname { get; set; }
    public int duration { get; set; }
    public Dictionary<string, List<string>> questions { get; set; }

    // EVENT DECLARATION
    public event EventHandler<examArgs> startexam;
    //     ↑     ↑                        ↑
```

```
    //      |       |                          └── Event name
    //      |       └──────────────────────────── Event type
    //      └──────────────────────────────────── Keyword

    // Constructor
    public exam(int id, string subjectname, int duration,
                Dictionary<string, List<string>> questions)
    {
        this.id = id;
        this.subjectname = subjectname;
        this.duration = duration;
        this.questions = questions;
    }

    // METHOD TO RAISE EVENT
    public void OnStartExam()
    {
        // Null-conditional: only invoke if subscribers exist
        startexam?.Invoke(
            this,  // sender: this exam instance
            new examArgs("lec3", DateTime.Now.AddMinutes(15))  // event data
        );
    }
}
```

Step 3: Subscriber Classes

```
// SUBSCRIBER 1: Student
class Student
{
    public int id { get; set; }
    public string name { get; set; }
    public int age { get; set; }

    // EVENT HANDLER METHOD
    // Signature MUST match EventHandler<examArgs>:
    // void MethodName(object sender, examArgs e)
    public void answerexam(object sender, examArgs e)
    {
        // Cast sender to get exam details
        exam ex = sender as exam;

        // Use exam and event data
        Console.WriteLine(ex);  // Calls ToString()
        Console.WriteLine($"student {name} start answer exam @{e.location}");
```

```
    }
}

// SUBSCRIBER 2: Admin
class Admin
{
    public int id { get; set; }
    public string name { get; set; }
    public int age { get; set; }

    // EVENT HANDLER METHOD
    public void monitior(object sender, examArgs e)
    {
        exam ex = sender as exam;

        Console.WriteLine($"admin {name} start monitor {ex.subjectname} " +
                          $"exam location {e.location} @ " +
                          $"{e.starttime.ToShortTimeString()} " +
                          $",exam duration: {ex.duration} Mintues ...");
    }
}
```

Step 4: Wire Everything Together

```
// Create questions dictionary
Dictionary<string, List<string>> questions= new Dictionary<string,
List<string>>();

questions.Add(
    "What is Encapsulation in OOP?",
    new List<string>()
    {
        "1- Hiding implementation details and exposing only necessary parts",
        "2- Creating multiple objects from one class",
        "3- Ability of object to take many forms",
        "4- Reusing code through inheritance"
    }
);
questions.Add(
    "Which keyword is used to inherit a class in C#?",
    new List<string>()
    {
        "1- extends",
        "2- implements",
        "3- inherits",
```

```
        "4- :"
    }
);

questions.Add(
    "What is the main purpose of a constructor?",
    new List<string>()
    {
        "1- To destroy objects",
        "2- To initialize object data",
        "3- To call private methods",
        "4- To override base methods"
    }
);

questions.Add(
    "What does 'this' keyword refer to?",
    new List<string>()
    {
        "1- The current object",
        "2- The parent class",
        "3- A static reference",
        "4- A global variable"
    }
);


// Create PUBLISHER
exam ex = new exam(1, "C#", 90, questions);

// Create SUBSCRIBER 1
Student st = new Student(2, "ali", 20);
ex.startexam += st.answerexam;  // Subscribe to event
//            ↑
//            └── Only += and -= allowed for events!

// Create SUBSCRIBER 2
Admin ad = new Admin(3, "mahmoud", 40);
ex.startexam += ad.monitior;  // Subscribe to event

// RAISE EVENT
ex.OnStartExam();
// This will call:
// 1. st.answerexam(ex, eventArgs)
// 2. ad.monitior(ex, eventArgs)
```

Important Notes

⚠ **Event Protection**

```
// ❌ These will NOT compile:
ex.startexam = st.answerexam;      // ERROR: Cannot assign
ex.startexam.Invoke(ex, args);     // ERROR: Cannot invoke
ex.startexam = null;               // ERROR: Cannot assign

// ✅ Only these work:
ex.startexam += st.answerexam;     // Subscribe
ex.startexam -= st.answerexam;     // Unsubscribe
```

Event Best Practices

1. Naming Conventions

```
// ✅ GOOD - Standard conventions
class ButtonClickEventArgs : EventArgs { }
public event EventHandler<ButtonClickEventArgs> ButtonClicked;
protected virtual void OnButtonClicked(ButtonClickEventArgs e) { }

// ❌ BAD - Non-standard
class clickdata { }  // Should be ClickEventArgs
public event Action onclick;  // Should be EventHandler<T>
public void fireclick() { }  // Should be OnButtonClicked
```

2. Always Check for Null

```
// ✅ GOOD - Null-conditional operator
protected virtual void OnStartExam(examArgs e)
{
    startexam?.Invoke(this, e);
}

// ❌ BAD - Can throw NullReferenceException
protected virtual void OnStartExam(examArgs e)
{
    startexam.Invoke(this, e);  // Crashes if no subscribers!
}
```

```
// ⚠️ OK but verbose
protected virtual void OnStartExam(examArgs e)
{
    if (startexam != null)
    {
        startexam.Invoke(this, e);
    }
}
```

3. Use Virtual Methods

```
// ✅ GOOD - Virtual allows derived classes to override
protected virtual void OnStartExam(examArgs e)
{
    startexam?.Invoke(this, e);
}

// Derived class can add logic
class AdvancedExam : exam
{
    protected override void OnStartExam(examArgs e)
    {
        // Custom logic before
        Console.WriteLine("Advanced exam starting...");

        // Call base to raise event
        base.OnStartExam(e);

        // Custom logic after
        Console.WriteLine("Advanced exam started!");
    }
}
```

4. Thread-Safe Event Raising

```
// ✅ GOOD - Thread-safe
protected virtual void OnStartExam(examArgs e)
{
    // Copy reference to avoid race condition
    EventHandler<examArgs> handler = startexam;
    handler?.Invoke(this, e);
}
```

```
// Or use C# 6.0+ null-conditional (also thread-safe)
protected virtual void OnStartExam(examArgs e)
{
    startexam?.Invoke(this, e);
}
```

5. Unsubscribe When Done

```
// ✅ GOOD - Clean up subscriptions
class Student
{
    public void StartListening(exam ex)
    {
        ex.startexam += answerexam;
    }

    public void StopListening(exam ex)
    {
        ex.startexam -= answerexam;  // Important!
    }
}

// Prevents memory leaks in long-running applications
```

6. Meaningful EventArgs

```
// ✅ GOOD - Descriptive properties
class ExamEventArgs : EventArgs
{
    public string Location { get; set; }
    public DateTime StartTime { get; set; }
    public TimeSpan Duration { get; set; }
    public string SubjectName { get; set; }
    public int TotalQuestions { get; set; }
}

// ❌ BAD - Generic names
class examArgs
{
    public string data1 { get; set; }
    public string data2 { get; set; }
}
```

Real-World Examples

Example 1: Download Progress Event

```
class DownloadProgressEventArgs : EventArgs
{
    public long BytesReceived { get; set; }
    public long TotalBytes { get; set; }
    public int ProgressPercentage { get; set; }
}

class Downloader
{
    public event EventHandler<DownloadProgressEventArgs> ProgressChanged;

    public void DownloadFile(string url)
    {
        long totalBytes = 1000000;

        for (long bytesReceived = 0; bytesReceived <= totalBytes;
bytesReceived += 10000)
        {
            // Raise progress event
            OnProgressChanged(new DownloadProgressEventArgs
            {
                BytesReceived = bytesReceived,
                TotalBytes = totalBytes,
                ProgressPercentage = (int)((bytesReceived * 100) / totalBytes)
            });

            Thread.Sleep(100);  // Simulate download
        }
    }

    protected virtual void OnProgressChanged(DownloadProgressEventArgs e)
    {
        ProgressChanged?.Invoke(this, e);
    }
}

// Usage
Downloader downloader = new Downloader();
downloader.ProgressChanged += (sender, e) =>
{
    Console.WriteLine($"Progress: {e.ProgressPercentage}% " +
                      $"({e.BytesReceived}/{e.TotalBytes} bytes)");
```

```
};
downloader.DownloadFile("http://example.com/file.zip");
```

Example 2: Stock Price Alert

```
class PriceChangedEventArgs : EventArgs
{
    public decimal OldPrice { get; set; }
    public decimal NewPrice { get; set; }
    public decimal ChangeAmount => NewPrice - OldPrice;
    public decimal ChangePercent => (ChangeAmount / OldPrice) * 100;
}

class Stock
{
    private decimal _price;
    public string Symbol { get; set; }

    public decimal Price
    {
        get => _price;
        set
        {
            if (_price != value)
            {
                decimal oldPrice = _price;
                _price = value;

                // Raise event when price changes
                OnPriceChanged(new PriceChangedEventArgs
                {
                    OldPrice = oldPrice,
                    NewPrice = value
                });
            }
        }
    }

    public event EventHandler<PriceChangedEventArgs> PriceChanged;

    protected virtual void OnPriceChanged(PriceChangedEventArgs e)
    {
        PriceChanged?.Invoke(this, e);
    }
}
```

```
// Usage
Stock stock = new Stock { Symbol = "AAPL", Price = 150.00m };

// Alert subscriber
stock.PriceChanged += (sender, e) =>
{
    Stock s = sender as Stock;
    Console.WriteLine($"{s.Symbol}: ${e.OldPrice} → ${e.NewPrice} " +
                      $"({e.ChangePercent:F2}%)");
};

stock.Price = 155.50m;  // Triggers event
// Output: AAPL: $150.00 → $155.50 (3.67%)
```

Example 3: Multi-Subscriber Logger

```
class LogEventArgs : EventArgs
{
    public string Message { get; set; }
    public DateTime Timestamp { get; set; }
    public string Level { get; set; }
}

class Logger
{
    public event EventHandler<LogEventArgs> LogWritten;

    public void Log(string message, string level = "INFO")
    {
        OnLogWritten(new LogEventArgs
        {
            Message = message,
            Timestamp = DateTime.Now,
            Level = level
        });
    }

    protected virtual void OnLogWritten(LogEventArgs e)
    {
        LogWritten?.Invoke(this, e);
    }
}

// Usage
```

```
Logger logger = new Logger();

// Console logger
logger.LogWritten += (sender, e) =>
{
    Console.WriteLine($"[{e.Timestamp:HH:mm:ss}] {e.Level}: {e.Message}");
};

// File logger
logger.LogWritten += (sender, e) =>
{
    File.AppendAllText("log.txt",
        $"[{e.Timestamp}] {e.Level}: {e.Message}\n");
};

// Email logger (for errors)
logger.LogWritten += (sender, e) =>
{
    if (e.Level == "ERROR")
    {
        // SendEmail(e.Message);
        Console.WriteLine($"Email sent about: {e.Message}");
    }
};

// All three loggers will be notified!
logger.Log("Application started");
logger.Log("Critical error occurred", "ERROR");
```

Summary

Key Concepts

✓ Events Summary

What are Events?

- Encapsulated multicast delegates
- Implement publisher-subscriber pattern
- Provide loose coupling between classes

Components:

1. **EventArgs** - Data container
2. **Publisher** - Class that raises event
3. **Subscribers** - Classes that handle event

Benefits:

- ✅ Loose coupling
- ✅ Easy to extend
- ✅ Multiple subscribers
- ✅ Protected from external misuse

When to Use:

- User interactions (clicks, input)
- State changes (property changed)
- Notifications (progress, completion)
- Monitoring (logging, alerts)

Pattern Template

```
// 1. EventArgs
class MyEventArgs : EventArgs
{
    public string Data { get; set; }
}

// 2. Publisher
class Publisher
{
    public event EventHandler<MyEventArgs> MyEvent;

    protected virtual void OnMyEvent(MyEventArgs e)
    {
        MyEvent?.Invoke(this, e);
    }

    public void DoSomething()
    {
        // ... do work ...
        OnMyEvent(new MyEventArgs { Data = "Something happened" });
    }
}
```

```
// 3. Subscriber
class Subscriber
{
    public void HandleEvent(object sender, MyEventArgs e)
    {
        Publisher pub = sender as Publisher;
        Console.WriteLine($"Received: {e.Data}");
    }
}

// 4. Usage
Publisher pub = new Publisher();
Subscriber sub = new Subscriber();
pub.MyEvent += sub.HandleEvent;
pub.DoSomething();  // Triggers event
```

End of Documentation

🖊 Key Takeaways

- **Events** are protected multicast delegates
- Use **EventHandler < T > for standard pattern
- **EventArgs** carries event data
- **object sender** identifies event source
- Always use **?.Invoke()** for null-safety
- Events enable **loose coupling**
- Perfect for **publish-subscribe** patterns

Abdullah Ali

Contact : +201012613453