Chapter 2: Core Event-Driven Programming

Ref. Murach's C# (7th Ed.), Ch. 4–7; Deitel & Deitel (2017), Ch. 4–6, 12; csharpkey.com (Events and Delegates)

Chapter 2: Core Event-Driven Programming

- Review of C# basics (variables, data types, control structures)
- Delegates and event handlers
- Event-driven design patterns
- Working with arrays, lists, and strings
- Applying OOP principles (classes, inheritance) in GUI apps

How to Code Control Structure

- Control structures allow programs to make decisions and repeat actions based on conditions.
- Like other programming language, C# support three main types common to modern programming languages: **selection**, **case**, and **iteration** structures.
- Key topics include Boolean expressions, conditional statements (if-else), switch statements and expressions, and the conditional operator.

- **Boolean Expressions:** Boolean expressions evaluate to true or false. They are essential for control structures because you need to learn how to code them before implementing control flow.
- **Relational Operators:** Relational operators compare two operands to create a Boolean expression. Work on literals, variables, arithmetic expressions, or keywords like null, true, or false.

Relational Operators		
Operator	Name	Description
==	Equality	Returns a true value if the left and right operands are equal.
!=	Inequality	Returns a true value if the left and right operands are not equal.
>	Greater than	Returns a true value if the left operand is greater than the right operand.
<	Less than	Returns a true value if the right operand is greater than the left operand (i.e., left is less than right).
>=	Greater than or equal	Returns a true value if the left operand is greater than or equal to the right operand.
<=	Less than or equal	Returns a true value if the left operand is less than or equal to the right operand.

- **Examples**
  - firstName == “Amin" // Equal to a string literal
  - txtYears.Text == "" // Equal to an empty string
  - message == null // Equal to a null value
  - isValid == false // Equal to the false value
  - code == productCode // Equal to another variable
  - lastName != “Gelila" // Not equal to a string literal
  - years > 0 // Greater than a numeric literal
  - i < months // Less than a variable
  - subtotal >= 500 // Greater than or equal to a literal value
  - quantity <= reorderPoint // Less than or equal to

Note this on Relational Expression

- Use relational operators to compare two operands and return a Boolean value.
- For equality, use two equals signs (==). A single = is for assignment.
- When comparing strings, use the equality and inequality operators. C# will cast less precise operands to the type of the more precise operand.
- Operands can be numeric, string, or other types, but they must be comparable (e.g., numeric with numeric).
- If your code won't compile due to incompatible types, the compiler will interpret it as an assignment statement instead.

- **Logical Operators:** Logical operators combine two or more Boolean expressions into a compound expression. They include **conditional-And (&&)**, **conditional-Or (||)**, **And (&)**, **Or (|)**, and **Not (!)**.

Examples

- subtotal > 250 && subtotal < 500 // Conditional-And
- timeInService <= 4 || timeInService > 12 // Conditional-Or
- isValid == true & counter++ < years // And (always evaluates both)
- isValid == true | counter++ < years // Or (always evaluates both)
- date > startDate && date < expirationDate || !isValid == true // Mixed
- (thisYear > lastYear) | (empType=="Part time") && startYear < currentYear // With parentheses
- !(counter++ > years) // Not

Logical operators

Operator	Name	Description
&&	Conditional-And	Returns a true value if both expressions are true. This operator only evaluates the second expression if necessary.
||	Conditional-Or	Returns a true value if either expression is true. This operator only evaluates the second expression if necessary.
&	And	Returns a true value if both expressions are true. This operator always evaluates both expressions.
|	Or	Returns a true value if either expression is true. This operator always evaluates both expressions.
!	Not	Reverses the value of the expression.

Note on Logical Operators

- Use logical operators to create compound Boolean expressions.
- The **&&** and **||** are short-circuit operators: They skip the second expression if the first determines the result (e.g., if the first is false in &&, the overall is false).
- The **&** and always evaluate both expressions, which is useful if the second has side effects (e.g., incrementing a variable).
- The **!** reverses a Boolean value.
- For more than two expressions, evaluation is left-to-right, but parentheses can change the order for clarity.
- Short-circuit operators are more efficient and common unless you need both expressions evaluated.

Conditional Statements and Expressions

- Conditional statements use Boolean expressions to select actions. This includes if-else for selection.
- The if-else statement (or just if statement) is the primary selection structure in C#. It lets you select different actions based on a Boolean expression.

Syntax

```
if (booleanExpression) { statements; }
else if (booleanExpression) { statements; } ...
else { statements; }
```

If without else

```
if (subtotal > 100)
    discountPercent = .2m;
```

If with else

```
if (subtotal > 100)
    discountPercent = .2m;
else
    discountPercent = .1m;
```

With block statement

```
if (subtotal > 100)
{
    status = "Bulk rate";
    discountPercent = .2m;
}
```

If with else if and else

```
if (subtotal >= 100 && subtotal < 200)
    discountPercent = .2m;
else if (subtotal >= 200 && subtotal < 300)
    discountPercent = .3m;
else if (subtotal >= 300)
    discountPercent = .4m;
else
    discountPercent = .1m;
```

Nested if else

```
if (customerType == "R")
{
    if (subtotal >= 100)
        discountPercent = .2m;
    else
        discountPercent = .1m;
}
else // customerType isn't "R"
    discountPercent = .4m;
```

Note on if else

- An if-else statement always contains an if clause. Else if and else are optional.
- Use braces {} for blocks with multiple statements or for clarity. A single statement doesn't need braces, but it's good practice.
- The expression must be Boolean. If true, executes the if block; if false, skips to else if/else or continues.
- Nested ifs allow more complex logic. Indent nested statements for readability.
- End with a semicolon only if the clause has a single statement without braces.
- Common in menus (e.g., Quick Actions in Visual Studio) or refactoring.
- Avoid dangling else by using braces.

Switch Statements: Switch statements handle case structures, allowing selection based on a match expression's value against case labels.

Syntax

```
switch (matchExpression)
{
    case constantExpression:
        statements;
        break;
    case constantExpression:
        statements;
        break;
    ...
    default:
        statements;
        break;
}
```

Example with default code

```
switch (customerType)
{
    case "R":
        discountPercent = .1m;
        break;
    case "C":
        discountPercent = .2m;
        break;
    default:
        discountPercent = .0m;
        break;
}
```

Note on Switch

- The match expression is evaluated (e.g., variable, literal, expression). It must be integral, string, char, bool, enum, or any integer type.
- Case labels are constants. Execution transfers to the matching case.
- Include a break; to exit the switch; otherwise, it falls through to the next case (useful for shared code).
- Default label (optional) handles non-matching values.
- No fall-through rule in C#: You must use break unless the case is empty.
- Useful when if-else would be lengthy with many equality checks.
- In C# 8.0+, switch statements are limited; use switch expressions for more flexibility.

- **Switch Expressions:** Switch expressions (C# 8.0+) are concise alternatives to switch statements, often assigning results to variables.

```
matchExpression switch
{
    constantExpression => expression,
    constantExpression => expression,
    ...
    _ => expression
};
```

```
discountPercent = customerType switch
{
    "R" => .1m,
    "C" => .2m,
    _ => .0m
};
```

Note

- Switch expressions evaluate the match and return a value (no break needed).
- Use _ (discard) for default.
- Supports patterns like relational operators (>= C# 9.0) and logical (and, or, not).
- More concise than switch statements; no fall-through by default.
- The lambda operator (=>) separates patterns from results.
- All paths must return a value; exhaustive.
- Useful for assigning values based on conditions without full statements.

- Reading assignment on loops..

Delegates, Event & Event Handlers

Delegates, Event & Event Handlers

- Delegates, events, and lambda expressions are foundational to event-driven programming in C#, enabling callbacks, notifications, and concise implementations.
- Delegates act as type-safe method references, supporting multicasting and forming the basis for events.
- Events implement the publisher-subscriber pattern, allowing objects to signal changes securely.
- Lambda expressions and anonymous methods provide inline, succinct ways to define behaviors.

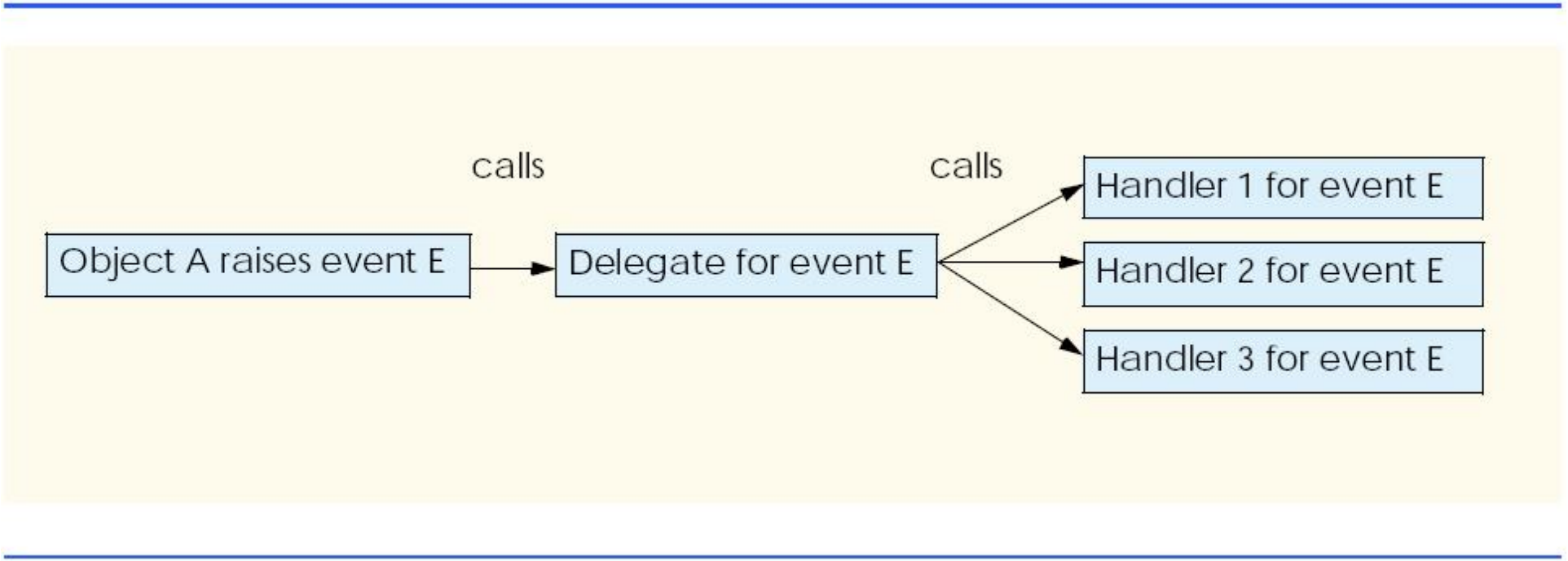
- Graphical User Interfaces (GUIs) in .NET operate in an **event-driven** manner, meaning they respond to user interactions by generating **events**. These interactions include actions like:
  - Clicking a button
  - Moving or clicking the mouse
  - Typing in a text box
  - Selecting a menu item
  - Closing a window
- **Events** are notifications that something has occurred, and **event handlers** are methods designed to process these events and perform specific tasks.
- For example, clicking a button might trigger an event that changes the background color of a form.

- **Events:** An **event** is a message sent by an object (e.g., a button or form) to signal an action, such as a user click. Events are typically predefined in .NET controls, and each control can raise specific events based on its functionality.
- **Delegates:** A **delegate** is a type-safe function pointer in .NET that defines the signature (return type and parameters) of methods it can reference. Delegates are critical in event handling because they act as intermediaries between the event source (e.g., a button) and the method that handles the event (the event handler).
- **Multicast Delegates**: Delegates in .NET are derived from the MulticastDelegate class, meaning they can hold references to multiple methods with the same signature. When an event is raised, all methods referenced by the delegate are invoked.
- Example: A button's Click event is associated with a delegate that specifies the signature of its event handlers.

- **Event Handlers:** An **event handler** is a method that processes an event. It must match the signature defined by the associated delegate. Typically, an event handler:
  - Has a void return type.
  - Takes two parameters:
    - sender: A reference to the object that raised the event (e.g., the button clicked).
    - e: An instance of EventArgs (or a derived class) containing event-specific data.

```
void ControlName_EventName(object sender, EventArgs e)
{
    // Event-handling logic here
}
```

Event handling mode



- Once an event is raised, every method that the delegate references is called.
- Every method in the delegate must have the same signature, because they are all passed the same information.

Basic Event Handling in .NET

- .NET controls like buttons, text boxes, and forms come with predefined events and delegates.
- Programmers can create event handlers and register them with these delegates to respond to user actions.
- Visual Studio simplifies this process by generating much of the required code automatically.
- **Steps to Create and Register an Event Handler**
  - **Create a Windows Forms Application**:
    - Open Visual Studio and create a new Windows Forms App (.NET Framework).
    - Add a control (e.g., a Button or Form) to the designer.
  - **Access the Events**:
    - In the **Properties** window of a control, click the **Events** icon (lightning bolt) to view available events.
    - Events like Click, MouseMove, or KeyPress are listed with descriptions.
  - **Generate an Event Handler**:
    - Double-click an event (e.g., Click) in the Properties window.
    - Visual Studio generates an empty event handler in the code with the correct signature:

Generate an Event Handler: Double-click an event (e.g., Click) in the Properties window.
Visual Studio generates an empty event handler in the code with the correct signature:

```
private void Form1_Click(object sender, EventArgs e)
{
    // Add event-handling code here
}
```

Add Event-Handling Logic: Insert code to perform the desired action. For example, to display a message box when a form is clicked:

```
private void Form1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Form is pressed.");
}
```

Register the Event Handler: Visual Studio automatically registers the event handler with the control's delegate in the InitializeComponent method:

```
this.Click += new System.EventHandler(this.Form1_Click);
```

- This code links the Click event of the form to the Form1_Click method using the System.EventHandler delegate.

- Delegates are classes referencing methods with matching signatures, allowing indirect invocation.
- **Key Characteristics**
  - **Delegate**: Type-safe method reference (signature + target). Enables callbacks, strategies; underlies events.
  - **Event**: Publisher-subscriber notification on a delegate. Subscribers(e.g button) add/remove handlers; only publisher raises.
  - **Event Handler**: Method matching event delegate, often (object sender, EventArgs e).
  - **Type-Safe**: Compile-time signature checks.
  - **Maintained Info**: Method address, parameters, return type.
  - **Multicasting**: Invoke multiple methods.
  - **Base Classes**: Derive from System.MulticastDelegate (from System.Delegate), with methods like Invoke(), GetInvocationList().
  - **Invocation**: Synchronous (Invoke()); async deprecated in modern .NET (use tasks).
- Delegates point to static/instance methods; immutable (operations create new instances).
- **Analogy**: Delegate = telephone number; Event = group broadcast; Handler = answering person.

Defining and Using Delegates

```
public delegate returnType DelegateName(parameterList);
```

```
public delegate int BinaryOp(int x, int y);
```

```
static int Add(int a, int b) => a + b;
BinaryOp op = Add;  // Method group conversion (preferred)
op = new BinaryOp(Add);  // Explicit
int sum = op(2, 3);  // Invoke
```

- **Passing as Parameters (Strategy/Callback, Deitel)**:

```
public static IEnumerable<int> Filter(int[] data, Predicate<int> test)
{
    foreach (var x in data) if (test(x)) yield return x;
}
var evens = Filter(nums, n => n % 2 == 0);
```

Anonymous Methods and Lambda Expressions

- **Anonymous Methods (Troelsen/Deitel)**: Inline, unnamed.

Inline, unnamed

```
op = delegate(int a, int b) { return a - b; };
```

Lambda Expressions: Concise with =>.

```
BinaryOp mul = (a, b) => a * b;
```

Expression-Bodied Members (C# 7.0+):

```
public int Add(int x, int y) => x + y;
```

Lambdas for short logic; named methods for complexity

Understanding Events

- Events encapsulate delegates, exposing only subscription.
- **Defining and Raising**

```
public event DelegateType EventName;
```

Standard Pattern

```
public class FileDownloader
{
    public event EventHandler Started;
    public event EventHandler<ProgressEventArgs> Progress;

    protected virtual void OnStarted() => Started?.Invoke(this, EventArgs.Empty);
    protected virtual void OnProgress(ProgressEventArgs e) => Progress?.Invoke(this, e);
}
public class ProgressEventArgs : EventArgs { public int Percent { get; } /* ... */ }
```

Subscribing and Unsubscribing

Generic EventHandler:

```
d.Progress += (s, e) => progressBar.Value = e.Percent;
d.Progress -= handler;  // Avoid leaks
```

Event Handling in WinForms

- **Designer Wiring**:
- Properties window (Events tab) auto-generates in InitializeComponent().
- Avoid editing .Designer.cs.
- **Manual Wiring**:

```
button.Click += Button_Click;
private void Button_Click(object sender, EventArgs e) { /* ... */ }
```

Shared Handlers

```
private void Option_CheckedChanged(object sender, EventArgs e)
{
    var cb = (CheckBox)sender;
    // Use sender/Tag to differentiate
}
```

Common Events:
EventHandler: Click, Load.
Specialized: MouseEventHandler, KeyEventHandler.

Event-Driven Design Patterns

Event-Driven Design Patterns

- Event-driven design patterns are essential for building responsive and scalable applications, particularly in GUI frameworks like WinForms, WPF, … within the .NET ecosystem.
- These patterns leverage delegates, events, and event handlers to manage asynchronous interactions and decouple components.

- **Delegate**: A type-safe method reference enabling callbacks and multicasting.
- **Event**: A publisher-subscriber mechanism built on delegates, raised by the publisher.
- **Event Handler**: A method (e.g., (object sender, EventArgs e)) responding to events.
- **Event-Driven GUI**: User actions (e.g., clicks) trigger events to update UI/state.

1. Observer Pattern

- The Observer pattern defines a one-to-many dependency between objects so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically.
- This promotes loose coupling, as observers don't need to poll the subject for changes.
- In .NET, it's often implemented using events and delegates, where the subject raises an event, and observers subscribe via event handlers.
- It's suitable for scenarios like data binding in GUIs, where UI elements update when data changes

Observer Pattern Explained:

- **Subject (Observable)**: Maintains a list of observers (via events) and notifies them of state changes.
- **Observer (Subscriber)**: Implements an update method (handler) to react to notifications.
- **Benefits**: Reduces direct dependencies; supports dynamic addition/removal of observers; aligns with .NET's event system for push-based notifications.
- **Drawbacks**: Can lead to memory leaks if observers aren't unsubscribed; over-notification if not filtered.
- **Use**: For real-time updates, like stock tickers or UI synchronization.

- **Practical Example**: A weather station (subject) notifies displays (observers) when temperature changes.

```
public class WeatherStation  // Subject
{
    public event EventHandler<TemperatureChangedEventArgs> TemperatureChanged;
    private double temperature;

    public double Temperature
    {
        get => temperature;
        set
        {
            temperature = value;
            TemperatureChanged?.Invoke(this, new TemperatureChangedEventArgs { NewTemperatur
        }
    }
}
```

```
public class TemperatureChangedEventArgs : EventArgs
{
    public double NewTemperature { get; init; }
}
```

```
public class Display  // Observer
{
    public Display(WeatherStation station)
    {
        station.TemperatureChanged += UpdateDisplay;
    }

    private void UpdateDisplay(object sender, TemperatureChangedEventArgs e)
    {
        Console.WriteLine($"Temperature updated to {e.NewTemperature}°C");
    }

    // Unsubscribe method for cleanup
    public void Unsubscribe(WeatherStation station)
    {
        station.TemperatureChanged -= UpdateDisplay;
    }
}
```

```
// Usage in GUI (e.g., WinForms button click sets temperature)
WeatherStation station = new();
Display display = new(station);
station.Temperature = 25.5;  // Notifies display
```

2. Publisher-Subscriber Pattern

- Also known as Pub-Sub, this pattern allows publishers(e.g.button) to send messages to multiple subscribers without knowing who they are, promoting decoupling.
- It's similar to Observer but often used in distributed systems (e.g., message queues).
- In .NET, it's implemented with events or libraries like MassTransit for pub-sub over buses.

3. Command Pattern

- Encapsulates a request as an object, allowing parameterization, queuing, and undo operations.
- Useful for GUI commands like menu actions.

4. Mediator Pattern

- Centralizes communication between objects (colleagues) via a mediator, reducing direct dependencies.
- Great for complex GUIs with interacting controls.
- **Benefits**: Simplifies maintenance; promotes single responsibility.
- **Drawbacks**: Mediator can become god object if not managed.
- **Use**: For chat apps, form validations, or Chat room where users communicate via mediator.

5. Asynchronous Event Pattern

- Defines asynchronous operations with events for progress/completion, allowing non-blocking GUI updates.
- In .NET, use EAP (Event-based Asynchronous Pattern) with methods like MethodAsync and events like MethodCompleted.
- **Async Method**: Starts operation.
- **Events**: ProgressChanged, Completed.
- **Benefits**: Keeps UI responsive; supports cancellation/progress.
- **Drawbacks**: Deprecated in favor of TAP (Task-based); use for legacy.
- **Use**: Background tasks like downloads.

Example: Async file download with progress.

```
public class Downloader
{
    public event AsyncCompletedEventHandler DownloadCompleted;
    public event ProgressChangedEventHandler DownloadProgressChanged;

    public void DownloadAsync(string url)
    {
        // Simulate async download
        Task.Run(() =>
        {
            for (int i = 0; i <= 100; i += 10)
            {
                Thread.Sleep(200);
                DownloadProgressChanged?.Invoke(this, new ProgressChangedEventArgs(i, null));
            }
            DownloadCompleted?.Invoke(this, new AsyncCompletedEventArgs(null, false, null));
        });
    }
}
```

```
// Usage in GUI
Downloader d = new();
d.DownloadProgressChanged += (s, e) => progressBar.Value = e.ProgressPercentage;
d.DownloadCompleted += (s, e) => label.Text = "Download complete";
d.DownloadAsync("https://example.com/file");
```

- End of Events, EventHandler, Design Pattern…