

UNIT 4

Advanced C#

(14 Hours) — DotNet Technology (CAC302)

This unit moves beyond the basics of C# into the language and framework features that make modern .NET development productive: delegates and events for building loosely coupled, notification-driven designs; lambda expressions as a concise way to write inline functions; structured exception handling for robust error recovery; LINQ for declarative, type-safe data querying; and, finally, practical application — using .NET to talk to databases and to build web applications with ASP.NET.

1. Delegates

A delegate is a type-safe object that holds a reference to one or more methods with a matching signature. Delegates are the foundation on which events, LINQ, and asynchronous callbacks are all built — they let you treat a method as data: store it in a variable, pass it as a parameter, or return it from another method.

- A delegate type is declared once, specifying the return type and parameter list any matching method must have.
- A delegate instance can point to a static method, an instance method, or (via a lambda) an anonymous block of code.
- Multicast delegates can hold references to more than one method; invoking the delegate calls every referenced method in the order they were added, using `+=`.
- Generic built-in delegate types – *Action*, *Func*, and *Predicate* – mean you rarely need to declare a custom delegate type yourself.

Example 4.1 — Declaring and Using a Delegate

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

static int Add(int a, int b) => a + b;
static int Multiply(int a, int b) => a * b;

Operation op = Add;
op += Multiply; // multicast – but return value keeps only the last result
int result = op(3, 4); // calls Add, then Multiply; result == 12
```

Built-in Delegate	Signature Shape	Typical Use
Action	void Method(...)	Perform an operation, no return value

Built-in Delegate	Signature Shape	Typical Use
Func	TResult Method(...)	Compute and return a value
Predicate	bool Method(T obj)	Test an object against a condition

Note: A delegate is essentially a type-safe function pointer with built-in multicast support — something C and C++ function pointers do not provide directly.

2. Events

An event is a language construct built on top of delegates that implements the publisher/subscriber pattern. A class that raises notifications (the publisher) exposes an event; other classes (subscribers) attach handler methods to it and get called automatically whenever the event is raised.

- Events restrict what a subscriber can do with the delegate: outside the declaring class, code may only += or -= a handler — it cannot invoke the event directly or replace the whole handler list, unlike a plain public delegate field.
- The standard .NET pattern uses *EventHandler* or *EventHandler<TEventArgs>* as the delegate type, with the signature *void Handler(object sender, TEventArgs e)*.
- Always check an event for null subscribers before raising it (or use the *?.Invoke* null-conditional operator) to avoid a *NullReferenceException* when nobody has subscribed.

Example 4.2 — Publisher/Subscriber with an Event

```
public class Button
{
    public event EventHandler Clicked;

    public void SimulateClick() => Clicked?.Invoke(this, EventArgs.Empty);
}

var button = new Button();
button.Clicked += (sender, e) => Console.WriteLine("Button was clicked!");
button.SimulateClick(); // prints: Button was clicked!
```

Note: Events are the mechanism behind almost every UI framework in .NET — Windows Forms, WPF, and ASP.NET all raise events (Click, TextChanged, Load, ...) that application code subscribes to.

3. Lambda Expressions

A lambda expression is a compact, inline way of writing an anonymous function. Lambdas are most often used to supply the target for a delegate parameter without declaring a separate named

method — for event handlers, LINQ queries, and callbacks.

Form	Syntax	Example
Expression lambda	(params) => expression	<code>x => x * x</code>
Statement lambda	(params) => { statements; }	<code>x => { Console.WriteLine(x); return x; }</code>
No parameters	() => expression	<code>() => DateTime.Now</code>
Multiple parameters	(a, b) => expression	<code>(a, b) => a + b</code>

- Lambdas capture variables from their enclosing scope by reference, forming a closure — the captured variable's current value is used whenever the lambda eventually runs, not the value at the moment it was written.
- The compiler infers parameter types from context wherever possible, so explicit types are usually unnecessary.
- Lambdas are commonly assigned to *Func* and *Action* delegates and are the standard way to write LINQ predicates and projections (see Section 5).

Example 4.3 — Lambdas as Delegate Targets

```
Func<int, int, int> add = (a, b) => a + b;  
Console.WriteLine(add(2, 3)); // 5  
  
List<int> numbers = new() { 1, 2, 3, 4, 5 };  
var evens = numbers.Where(n => n % 2 == 0).ToList(); // { 2, 4 }
```

*Note: Because a lambda is just syntax for creating a delegate instance (or an expression tree, when the target is *IQueryable/Expression<T>*), everything said about delegates in Section 1 applies to lambdas too.*

4. Exception Handling

C# uses structured exception handling to separate normal program logic from error-handling logic. When something goes wrong, code throws an exception object; execution unwinds the call stack until a matching *catch* block is found, or the program terminates if none is.

Example 4.4 — try / catch / finally

```
try
{
    int[] numbers = { 1, 2, 3 };
    Console.WriteLine(numbers[5]); // throws IndexOutOfRangeException
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine("Index error: " + ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine("Unexpected error: " + ex.Message);
}
finally
{
    Console.WriteLine("Cleanup always runs here.");
}
```

- *catch* blocks are checked top-to-bottom, so more specific exception types must be listed before more general ones (e.g. before the base *Exception* class).
- *finally* always executes, whether or not an exception was thrown or caught — ideal for releasing resources.
- Use *throw* to raise a new exception, and *throw;* (with no argument, inside a catch block) to re-throw the current exception without resetting its stack trace.
- Custom exception types should derive from *Exception* (or a suitable existing subclass) and are used when built-in exception types don't describe the failure precisely enough.

Common Exception	Typical Cause
NullReferenceException	Dereferencing a null object reference
IndexOutOfRangeException	Accessing an array with an invalid index
InvalidOperationException	Calling a method when the object is in a bad state
ArgumentException / ArgumentNullException	Invalid or missing method argument
DivideByZeroException	Integer division by zero

Note: The using statement introduced in Unit 1 (Section 3) is effectively syntactic sugar for a try/finally block that calls Dispose — exception handling and resource cleanup are closely related.

5. Introduction of LINQ

LINQ (Language Integrated Query) brings query capabilities directly into C# syntax, letting you filter, sort, group, and project data from in-memory collections, XML, and databases using a single, consistent, strongly-typed syntax instead of separate loops or query languages for each data source.

- LINQ works over any type that implements *IEnumerable<T>* (in-memory LINQ to Objects) or *IQueryable<T>* (translated to another query language, e.g. SQL, via LINQ to Entities/EF).
- Two equivalent syntaxes exist: query syntax (SQL-like keywords: from, where, select) and method syntax (extension methods with lambdas: Where, Select, OrderBy). Query syntax is compiled into method syntax by the compiler.
- Common operators include *Where* (filter), *Select* (project/transform), *OrderBy/OrderByDescending* (sort), *GroupBy* (group), and *First/FirstOrDefault/Count/Sum* (aggregation).
- LINQ queries are lazily evaluated – a query variable just describes the query; the actual work happens when the results are enumerated (e.g. in a foreach loop or a call to *ToList()*).

Example 4.5 — Query Syntax vs Method Syntax

```
// Query syntax
var adultsQuery =
    from p in people
    where p.Age >= 18
    orderby p.Name
    select p.Name;

// Equivalent method syntax
var adultsMethod = people
    .Where(p => p.Age >= 18)
    .OrderBy(p => p.Name)
    .Select(p => p.Name);
```

Note: LINQ variants include LINQ to Objects (collections in memory), LINQ to XML (querying XDocument/XElement trees), and LINQ to Entities (querying a database through Entity Framework) — the same operators and syntax apply across all of them.

6. Working with Databases

.NET provides two complementary layers for talking to relational databases: the lower-level ADO.NET API, and the higher-level Entity Framework (EF) / EF Core object-relational mapper (ORM) built on top of it.

6.1 ADO.NET

- *SqlConnection* – opens and manages a connection to the database.
- *SqlCommand* – represents a SQL statement or stored-procedure call to execute against a connection.
- *SqlDataReader* – provides fast, forward-only, read-only streaming access to query results.
- *DataSet* / *DataTable* – an in-memory, disconnected cache of data that can be modified offline and later synchronised back to the database.

Example 4.6 — Querying with ADO.NET

```
using var connection = new SqlConnection(connectionString);
connection.Open();

using var command = new SqlCommand("SELECT Id, Name FROM Students", connection);
using var reader = command.ExecuteReader();
while (reader.Read())
{
    Console.WriteLine($"{reader["Id"]}: {reader["Name"]}");
}
```

6.2 Entity Framework (Core)

- EF Core maps C# classes (entities) directly to database tables, letting developers query and update data using LINQ instead of writing raw SQL.
- A *DbContext* subclass represents a session with the database and exposes *DbSet<T>* properties for each entity type.
- EF Core supports Code First (classes generate the schema via migrations), Database First (classes are scaffolded from an existing schema), and change tracking so that *SaveChanges()* automatically issues the correct INSERT/UPDATE/DELETE statements.

Worked Example 4.7 — Querying with EF Core and LINQ

Problem: Retrieve the names of all students older than 18, ordered alphabetically, using EF Core.

Solution:

```
public class SchoolContext : DbContext
{
    public DbSet<Student> Students { get; set; }
}

using var context = new SchoolContext();
var names = context.Students
    .Where(s => s.Age > 18)
    .OrderBy(s => s.Name)
    .Select(s => s.Name)
    .ToList();
```

EF Core translates this LINQ query into an equivalent SQL SELECT statement, sends it to the database, and materialises the results back into a List<string> — the developer never writes SQL directly.

Aspect	ADO.NET	Entity Framework Core
Abstraction level	Low – raw SQL and readers	High – classes/LINQ, ORM
Query language	Raw SQL strings	LINQ, translated to SQL
Performance control	Very fine-grained	Slightly less direct control
Development speed	Slower, more boilerplate	Faster, less boilerplate

7. Writing Web Applications using ASP.NET

ASP.NET (and its modern, cross-platform successor ASP.NET Core) is Microsoft's framework for building web applications and services in C#. It provides the request-handling pipeline, templating, routing, and hosting infrastructure needed to turn C# code into HTTP responses.

Model	Description
ASP.NET Web Forms	Legacy, event-driven, page-based model similar to Windows Forms
ASP.NET MVC	Model-View-Controller separation of concerns; controllers handle requests
ASP.NET Core Razor Pages	Page-focused model with C# and HTML mixed via Razor syntax
ASP.NET Core Web API	Framework for building HTTP/REST services returning JSON/XML

- The MVC pattern separates an application into the Model (data and business logic), the View (Razor templates that render HTML), and the Controller (handles incoming requests, talks to the model, and chooses a view).
- Razor syntax (@) lets C# code be embedded directly inside HTML markup to generate dynamic pages.
- Routing maps incoming URL patterns to specific controller actions or Razor pages, e.g. */Students/Details/5* routes to a *Details* action with *id = 5*.
- ASP.NET Core Web API controllers return data (often JSON) rather than HTML, making them the standard way to build backends for single-page applications and mobile apps.

Example 4.8 — A Minimal ASP.NET Core Controller

```
[ApiController]
[Route("api/students")]
public class StudentsController : ControllerBase
{
    private readonly SchoolContext _context;

    public StudentsController(SchoolContext context) => _context = context;

    [HttpGet]
    public IActionResult GetAll() =>
        Ok(_context.Students.ToList());
}
```

Note: This one small controller ties together almost everything in this unit: LINQ (ToList query), EF Core (SchoolContext, Section 6), and ASP.NET Core routing/attributes — illustrating how the individual topics of Unit 4 combine in a real application.

Key Takeaways

- ✓ Delegates are type-safe references to methods and are the building block for events, lambdas, and callbacks.
- ✓ Events use delegates to implement a controlled publisher/subscriber pattern, restricting subscribers to += / -=.
- ✓ Lambda expressions are concise inline functions, commonly used as delegate targets and in LINQ queries; they capture enclosing variables as closures.
- ✓ Structured exception handling (try/catch/finally, throw) separates error recovery from normal logic and guarantees cleanup code runs.
- ✓ LINQ provides a single, strongly-typed query syntax (query or method syntax) over in-memory collections, XML, and databases, and is lazily evaluated.
- ✓ ADO.NET gives fine-grained, low-level database access, while Entity Framework Core maps entities to tables and lets LINQ queries be translated automatically into SQL.
- ✓ ASP.NET Core supports multiple web programming models (MVC, Razor Pages, Web API), built around routing, controllers, and Razor templating.

— *End of Unit 4* —