DotNet Technology — CACS302

Last-Hour Exam Cheat Sheet — Units 1 to 4

UNIT 1 — Introducing C# and the .NET Framework (7 Hrs)

Object Orientation

- C# is a general-purpose, type-safe, object-oriented language.
- Core OOP pillars: Encapsulation, Inheritance, Polymorphism, Abstraction.
- Everything derives ultimately from System.Object — supports unification of the type system (value + reference types).

Type Safety

- C# is strongly typed and type-safe: the compiler/runtime ensures types are used consistently, preventing memory-corrupting casts.
- Static typing: variable types are known at compile time (most C# code).
- Dynamic typing: type resolved at runtime via the `dynamic` keyword (uses DLR — Dynamic Language Runtime).

Memory Management

- The CLR's Garbage Collector (GC) automatically reclaims memory of objects no longer referenced — programmer does not manually free memory.
- Managed heap: reference-type objects live here; GC compacts and reclaims.
- Stack: stores value types and method call frames; automatically popped on return.
- `IDisposable` / `using` statement — deterministic cleanup of unmanaged resources (files, connections).

Platform Support

- .NET runs across platforms via different implementations: .NET Framework (Windows-only, legacy), .NET Core / modern .NET (cross-platform: Windows, Linux, macOS), and Mono/Xamarin (mobile).

C# and the CLR (Common Language Runtime)

- C# compiles to IL (Intermediate Language) + metadata, packaged in an assembly (.dll/.exe).
- At runtime, the CLR uses a JIT (Just-In-Time) compiler to convert IL to native machine code.
- Because it targets the CLR, C# gets automatic memory management, type safety, exception handling, and interoperability with other CLR languages (VB.NET, F#).

CLR and the .NET Framework

- The .NET Framework = CLR (execution engine) + BCL/FCL (Base/Framework Class Library — collections, I/O, networking, etc.).
- Common Type System (CTS): defines how types are declared and used across languages.
- Common Language Specification (CLS): a subset of CTS rules that ensures cross-language compatibility.

Other Frameworks

- .NET Core — open-source, cross-platform, high-performance successor; basis for modern unified .NET (5/6/7/8+).
- Xamarin / .NET MAUI — cross-platform mobile and desktop app development.
- Mono — open-source implementation enabling C# on Linux/macOS/mobile before .NET Core existed.

Framework Overview

Layer	Role
Languages (C#, F#, VB.NET)	Source code compiled to IL
CLR	Executes IL; handles GC, JIT, type safety, exceptions
BCL / FCL	Reusable classes: collections, IO, threading, LINQ, networking
ASP.NET / WPF / WinForms / MAUI	Application frameworks built atop the BCL

.NET Standard 2.0

- .NET Standard is a formal specification of APIs that all .NET implementations (Framework, Core, Xamarin, Mono) agree to implement.
- Writing a library against .NET Standard 2.0 makes it usable from any compliant implementation — solves cross-platform code-sharing problems.
- Superseded going forward by simply targeting modern .NET (5+) with the unified BCL, but still relevant for older library compatibility.

Applied Technologies (built on .NET)

Technology	Purpose
ASP.NET / ASP.NET Core	Web applications and Web APIs
ADO.NET / Entity Framework (EF)	Database access (raw + ORM)
WPF / WinForms	Windows desktop GUI applications
.NET MAUI / Xamarin	Cross-platform mobile & desktop apps
LINQ	Unified query syntax over collections, XML, databases
WCF / gRPC / Web API	Service-oriented / distributed communication

Unit 1 — Quick Recall

✓ C# is type-safe, OOP, compiles to IL, executed by the CLR via JIT.
✓ GC manages memory automatically on the managed heap; stack holds value types.
✓ .NET Framework = CLR + BCL; CTS/CLS enable cross-language compatibility.
✓ .NET Standard 2.0 = common API surface across all .NET implementations.
✓ Applied tech: ASP.NET (web), EF/ADO.NET (data), WPF/MAUI (UI), LINQ (queries).

UNIT 2 — The C# Language Basics (12 Hrs)

Console and GUI Applications

- Console app: entry point is `static void Main(string[] args)` (or top-level statements in modern C#); I/O via `Console.WriteLine` / `Console.ReadLine`.
- GUI app (WPF/WinForms/MAUI): event-driven; a message loop dispatches UI events (clicks, input) to handler methods.

Minimal console app

```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Identifiers and Keywords

- Identifier: a name for a variable, method, class, etc. — must start with a letter or underscore, then letters/digits/underscores; case-sensitive.
- Keywords are reserved words (e.g. `class`, `if`, `return`) and cannot be used as identifiers unless prefixed with `@` (verbatim identifier, e.g. `@class`).

Writing Comments

Style	Syntax
Single-line	`// comment text`

Style	Syntax
Multi-line	`/* comment text */`
XML doc comment	`/// <summary>...</summary>`

Data Types

- Value types (stored on stack/inline): `int`, `long`, `short`, `byte`, `float`, `double`, `decimal`, `bool`, `char`, `struct`, `enum`.
- Reference types (stored on heap; variable holds a reference): `class`, `string`, `object`, `array`, `interface`, `delegate`.
- Predefined type aliases map to BCL structs, e.g. `int` = `System.Int32`, `string` = `System.String`.
- var — implicitly typed local variable; type inferred at compile time (still statically typed).
- Nullable value types: `int? x = null;` — wraps a value type to allow null.

Type	Size	Range/Notes
`bool`	1 byte	true / false
`char`	2 bytes	single UTF-16 character
`int`	4 bytes	-2,147,483,648 to 2,147,483,647
`long`	8 bytes	large integers, suffix L
`float`	4 bytes	single precision, suffix F
`double`	8 bytes	double precision (default for decimals)
`decimal`	16 bytes	high precision, suffix M — used for money
`string`	reference	immutable sequence of chars

Expressions and Operators

Category	Operators
Arithmetic	`+ - * / %`
Assignment	`= += -= *= /= %=`
Comparison	`== != < > <= >=`
Logical	`&& || !`
Bitwise	`& | ^ ~ << >>`
Null-related	`?? (null-coalescing) ?. (null-conditional)`
Ternary	`condition ? exprTrue : exprFalse`
Increment/Decrement	`++ -- (prefix and postfix)`

Strings and Characters

- `string` is immutable — every modification creates a new string; use `StringBuilder` for heavy concatenation in loops.
- String interpolation: `$"Hello {name}, you are {age}"`
- Verbatim string: `@"C:\path\no\escapes"`
- Common methods: `.Length`, `.Substring()`, `.Split()`, `.Trim()`, `.Replace()`, `.ToUpper()`, `.Contains()`

Arrays

Array declaration & usage

```
int[] nums = { 1, 2, 3, 4 };            // single-dimensional
int[,] grid = new int[3, 3];            // multi-dimensional (rectangular)
int[][] jagged = new int[3][];          // jagged array (array of arrays)
foreach (int n in nums) Console.WriteLine(n);
```

- Arrays are fixed-length, zero-indexed, reference types; `.Length` gives element count.

Variables and Parameters

Parameter modifier	Meaning
`(none)`	Passed by value — a copy is passed
`ref`	Passed by reference — must be initialized before the call
`out`	Passed by reference — must be assigned inside the method
`params`	Variable number of arguments collected into an array
`in`	Passed by reference, read-only inside the method

Statements

- Declaration: `int x = 5;`
- Expression statement: a method call or assignment ending in `;` e.g. `Console.WriteLine(x);`
- Selection statements: `if / else, switch`
- Iteration statements: `for, foreach, while, do...while`
- Jump statements: `break, continue, return, goto, throw`

Selection & iteration examples

```
if (x > 0) { ... } else if (x == 0) { ... } else { ... }
switch (day)
{
    case 1: Console.WriteLine("Mon"); break;
    default: Console.WriteLine("Other"); break;
}
for (int i = 0; i < 10; i++) { ... }
foreach (var item in list) { ... }
while (cond) { ... }
do { ... } while (cond);
```


Namespaces

- A namespace organizes types and avoids naming collisions: `namespace MyApp.Services { ... }`
- `using MyApp.Services;` imports a namespace so its types can be used unqualified.
- Modern C# supports file-scoped namespaces: `namespace MyApp.Services;` (no braces, applies to whole file).

Unit 2 — Quick Recall

✓ Value types → stack; reference types → heap. var infers type at compile time.
✓ string is immutable; use StringBuilder for repeated concatenation.
✓ Parameter modifiers: (value), ref, out, in, params.
✓ Statement categories: declaration, expression, selection, iteration, jump.
✓ Namespaces organize types and prevent name collisions; imported via using.

UNIT 3 — Creating Types in C# (12 Hrs)

Classes, Constructors and Deconstructors

Class with constructor & deconstructor

```
class Person
{
    public string Name;
    public int Age;
    public Person(string name, int age)      // constructor
    {
        Name = name; Age = age;
    }
    public void Deconstruct(out string name, out int age)   // deconstructor
    {
        name = Name; age = Age;
    }
}
// usage: var (n, a) = new Person("Ram", 20);
```

- A constructor initializes a new object; matches the class name, no return type. Multiple constructors can be overloaded.
- A deconstructor (via a `Deconstruct` method) lets an object be unpacked into separate variables.

The this Reference

- `this` refers to the current instance — used to disambiguate fields from parameters of the same name, or to chain constructors (`: this(...)`).

Properties

Auto-property & full property

```
public string Name { get; set; }              // auto-property
public string Name { get; private set; }       // restricted setter
private int _age;
public int Age                                 // full property
{
    get => _age;
    set { if (value >= 0) _age = value; }
}
```

Indexers

Indexer syntax

```
public class SampleList
{
    private string[] items = new string[10];
    public string this[int i]
    {
        get => items[i];
        set => items[i] = value;
    }
}
// usage: var x = sampleList[2];
```

- An indexer lets an object be accessed like an array using `obj[index]` syntax.

Static Constructors and Classes

- A static constructor runs once, before the first instance is created or any static member is accessed — has no access modifier or parameters.
- A static class (`static class Utils { }`) cannot be instantiated and can contain only static members.

Finalizers

- A finalizer (`~ClassName() { }`) is called by the GC before reclaiming an object's memory — used rarely, for releasing unmanaged resources; prefer `IDisposable`.

Dynamic Binding

- `dynamic x = GetValue();` defers type checking to runtime — useful for COM interop, reflection, or dynamic languages.
- Trade-off: no compile-time type checking or IntelliSense; errors surface only at runtime.

Operator Overloading

Overloading + for a custom type

```
public static Point operator +(Point a, Point b)
    => new Point(a.X + b.X, a.Y + b.Y);
```

- Lets custom types support standard operators (`+ - * / == !=` etc.) using natural syntax.

Inheritance

Basic inheritance

```
class Animal
{
    public virtual void Speak() => Console.WriteLine("...");
}
class Dog : Animal
{
    public override void Speak() => Console.WriteLine("Woof");
}
```

- C# supports single inheritance of classes (one base class) but multiple interface implementation.
- `virtual` marks a base method as overridable; `override` replaces it in a derived class; `sealed` prevents further overriding.

Abstract Classes and Methods, base Keyword

- An abstract class (`abstract class Shape { }`) cannot be instantiated directly and may contain abstract methods (no body — must be implemented by derived classes).
- `base.Method()` calls the base class's version of a method; `: base(args)` calls a base constructor.

Overloading

- Method overloading = multiple methods with the same name but different parameter lists (number/type of parameters) — resolved at compile time.

The Object Type

- `System.Object` (alias `object`) is the ultimate base class of every type in C# (value and reference).
- Key members: `ToString(), Equals(), GetHashCode(), GetType()`
- Boxing: converting a value type to `object` (heap-allocated wrapper). Unboxing: extracting the value type back — requires an explicit cast.

Structs

- A `struct` is a value type — good for small, immutable data (e.g. a Point). Copied by value; cannot inherit from another struct/class (implicitly derives from `System.ValueType`) but can implement interfaces.

Access Modifiers

Modifier	Visibility
`public`	Accessible from anywhere
`private`	Accessible only within the containing type (default for members)
`protected`	Accessible within the type and derived types
`internal`	Accessible within the same assembly
`protected internal`	protected OR internal (union)
`private protected`	protected AND internal (intersection)

Interfaces

Interface definition & implementation

```
interface IShape
{
    double Area();
}
class Circle : IShape
{
    public double Radius;
    public double Area() => Math.PI * Radius * Radius;
}
```

- An interface defines a contract (members with no implementation, by default) that implementing classes/structs must fulfil. A class can implement multiple interfaces.

Enums

Enum definition

```
enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }
Day today = Day.Monday;
```

- An `enum` defines a set of named integer constants; improves readability over magic numbers.

Generics

Generic class & method

```
class Box<T>
{
    public T Value;
    public Box(T value) { Value = value; }
}
Box<int> b = new Box<int>(5);
static T Max<T>(T a, T b) where T : IComparable<T>
    => a.CompareTo(b) > 0 ? a : b;
```

- Generics let types/methods be parameterized by type, giving type safety and reuse without boxing/casting. Constraints (`where T : ...`) restrict what T can be.

Unit 3 — Quick Recall

✓ Constructors initialize objects; this refers to the current instance; base accesses the parent class.

✓ Properties (get/set) and indexers give controlled, array-like access to data.

✓ Inheritance is single for classes, multiple for interfaces; virtual/override enable polymorphism; abstract classes can't be instantiated.

✓ object is the root type; boxing/unboxing convert between value types and object.

✓ Access modifiers control visibility: public, private, protected, internal, and their combinations.

✓ Generics (Box) give compile-time type safety without sacrificing reusability.

UNIT 4 — Advanced C# (14 Hrs)

Delegates

Delegate declaration & use

```
delegate int Operation(int a, int b);
static int Add(int a, int b) => a + b;
Operation op = Add;
int result = op(3, 4);      // result = 7
```

- A delegate is a type-safe function pointer — it holds a reference to one or more methods matching its signature and can invoke them.

- Built-in generic delegates: `Func<T, TResult>` (returns a value), `Action<T>` (returns void), `Predicate<T>` (returns bool).
- Multicast delegates: combine methods with +=; all are invoked in order when the delegate is called.

Events

Event declaration & subscription

```
class Button
{
    public event EventHandler Clicked;
    public void RaiseClick() => Clicked?.Invoke(this, EventArgs.Empty);
}
var btn = new Button();
btn.Clicked += (s, e) => Console.WriteLine("Clicked!");
btn.RaiseClick();
```

- An event is a controlled wrapper around a delegate — external code can only += / −= subscribe/unsubscribe, not directly invoke or overwrite it (encapsulation of the publish/subscribe pattern).
- Always check for null subscribers before raising: `Clicked?.Invoke(...)`

Lambda Expressions

Lambda syntax

```
Func<int, int, int> add = (a, b) => a + b;
Action<string> print = msg => Console.WriteLine(msg);
Predicate<int> isEven = n => n % 2 == 0;
var evens = numbers.Where(n => n % 2 == 0).ToList();   // LINQ + lambda
```

- A lambda expression (`params => expression/body`) is a concise, inline anonymous function — commonly used with delegates and LINQ.
- Lambdas can capture variables from their enclosing scope (closures).

Exception Handling

try / catch / finally

```
try
{
    int x = int.Parse(input);
    int result = 10 / x;
}
catch (FormatException ex)
{
    Console.WriteLine("Invalid number: " + ex.Message);
}
catch (DivideByZeroException)
{
    Console.WriteLine("Cannot divide by zero.");
}
finally
{
    Console.WriteLine("Cleanup runs regardless of outcome.");
}
```

- `try` wraps code that might throw; `catch` handles a specific exception type (most specific first); `finally` always runs (cleanup).
- `throw;` re-throws the caught exception preserving its stack trace; `throw new Exception(...)` throws a new one.
- Custom exceptions derive from `Exception` (or a more specific base) for domain-specific error types.

Introduction to LINQ (Language Integrated Query)

LINQ — method syntax vs query syntax

```
// Method syntax
var result = numbers.Where(n => n > 5)
                    .OrderBy(n => n)
                    .Select(n => n * 2);
// Query syntax
var result2 = from n in numbers
              where n > 5
              orderby n
              select n * 2;
```

- LINQ provides a unified query syntax across collections (`IEnumerable<T>`), XML, and databases (`IQueryable<T>` via EF).
- Common operators: `Where, Select, OrderBy, GroupBy, Join, Sum, Count, First, Any, ToList`
- LINQ queries are often lazily evaluated — the query runs only when enumerated (e.g. via `foreach` or `.ToList()`).

Working with Databases

- ADO.NET — lower-level data access using `SqlConnection, SqlCommand, SqlDataReader` to run raw SQL against a database.
- Entity Framework (EF) Core — an ORM (Object-Relational Mapper) that maps C# classes to database tables, letting you query with LINQ instead of raw SQL.

ADO.NET vs EF Core example

```
// ADO.NET
using var conn = new SqlConnection(connString);
conn.Open();
using var cmd = new SqlCommand("SELECT * FROM Students", conn);
using var reader = cmd.ExecuteReader();
while (reader.Read()) { Console.WriteLine(reader["Name"]); }
// EF Core
public class Student { public int Id; public string Name; }
public class AppDbContext : DbContext
{
    public DbSet<Student> Students { get; set; }
}
var students = dbContext.Students.Where(s => s.Id > 0).ToList();
```


- EF Core supports Code-First (classes generate the schema via migrations) and Database-First (schema generates the classes) approaches.

Writing Web Applications Using ASP.NET

- ASP.NET Core MVC: Model-View-Controller pattern — `Model` (data), `View` (Razor .cshtml UI), `Controller` (handles requests, returns responses).
- ASP.NET Core Web API: builds HTTP services returning JSON/XML, typically consumed by front-end apps or other services.
- Razor Pages: a page-focused alternative to MVC, combining a .cshtml page with a code-behind model.

Minimal ASP.NET Core Web API controller

```
[ApiController]
[Route("api/[controller]")]
public class StudentsController : ControllerBase
{
    private readonly AppDbContext _db;
    public StudentsController(AppDbContext db) => _db = db;
    [HttpGet]
    public IActionResult GetAll() => Ok(_db.Students.ToList());
    [HttpPost]
    public IActionResult Create(Student s)
    {
        _db.Students.Add(s);
        _db.SaveChanges();
        return CreatedAtAction(nameof(GetAll), s);
    }
}
```

- Dependency Injection (DI) is built into ASP.NET Core — services (like a DbContext) are registered in `Program.cs` and injected into controllers via constructors.
- Common HTTP method attributes: `[HttpGet]`, `[HttpPost]`, `[HttpPut]`, `[HttpDelete]`

Unit 4 — Quick Recall

✓ Delegates = type-safe function pointers; Func/Action/Predicate are built-in generic delegates.

✓ Events wrap delegates for safe publish/subscribe (+= / -=); always null-check before invoking.

✓ Lambdas (=>) are concise anonymous functions, heavily used with LINQ.

✓ try/catch/finally handles exceptions; catch specific types before general ones.

✓ LINQ unifies querying over collections and databases; method syntax vs query syntax produce the same result.

✓ ADO.NET = manual SQL access; EF Core = ORM mapping classes to tables via LINQ.

✓ ASP.NET Core: MVC (Model-View-Controller), Web API (JSON services), Razor Pages — all use built-in Dependency Injection.

— End of Cheat Sheet — Good luck! —