C#/.NET Learning Notes

Compiled for offline study and printing. Start with the Study Guide.

0

Study Guide: C#/.NET Exam Prep

Use this as your roadmap. Tiers reflect priority: Tier 1 first, then Tier 2, then Tier 3.

How to study

- Read the theory, then type out the examples yourself.
- After each section, answer the “Check yourself” questions without looking.
- Spaced repetition: revisit weak spots after 1–2 days.
- Practice: small katas; then build a tiny app that touches multiple topics.

Suggested sequence (2–3 weeks)

1. Tier 1 (Days 1–7): C# Basics, Flow, .NET ecosystem, OOP, Collections, Exceptions/Debugging.
2. Tier 2 (Days 8–13): Delegates/Events, LINQ, Async, ADO.NET/EF Core, File I/O, WPF basics.
3. Tier 3 (Days 14–18): ASP.NET Core, Blazor, Security, Mobile/Xamarin/MAUI, DevOps.

Check yourself (sample prompts)

- Explain value vs reference semantics; show a bug that arises from misunderstanding them.
- When would you use yield? Show a lazy pipeline over a large file.
- Demonstrate inheritance vs composition; when is each preferable?
- Write a LINQ query for: top 3 items per group; inner join vs group join difference.
- Show async/await with cancellation and explain why async void is dangerous.
- ADO.NET vs EF Core: trade-offs and when to choose each.
- WPF binding modes and validation: set up TwoWay binding with validation.

Good luck—keep it small, steady, and hands-on.

0

C# Basics: Data Types (Primitive, Value vs Reference)

What are Data Types?

Data types define the kind of data a variable can hold in a programming language. In C#, data types are crucial because they determine how much memory is allocated and what operations can be performed on the data.

Categories of Data Types in C#

1. Primitive (Built-in) Types: These are basic types provided by the language, such as int, double, char, and bool.
2. Value Types: These types store data directly. Examples include all primitive types (except string), structs, and enums. Value types are usually stored on the stack.
3. Reference Types: These types store a reference (address) to the actual data. Examples include string, arrays, classes, and delegates. Reference types are stored on the heap, and variables hold a reference to the memory location.

Value vs Reference Types

- Value Types: When you assign a value type variable to another, a copy of the value is made. Changes to one variable do not affect the other.
- Reference Types: When you assign a reference type variable to another, both variables refer to the same object in memory. Changes to one variable affect the other.

Why is this important?

Understanding the difference helps you predict how your data will behave when passed to methods or assigned to new variables, which is essential for writing bug-free code.

Examples

Value copy vs reference sharing:

0

```
// Value types: copy the value
int a = 42;
int b = a;    // copy
b++;
// a == 42, b == 43

// Reference types: copy the reference
int[] arr1 = { 1, 2, 3 };
int[] arr2 = arr1;  // same reference
arr2[0] = 99;
// arr1[0] == 99 and arr2[0] == 99

// Strings are reference types but immutable
string s1 = "hello";
string s2 = s1;
s2 = s2.ToUpperInvariant();
// s1 == "hello" (unchanged), s2 == "HELLO"
```

Tip: prefer small, immutable structs for simple data; use classes for entities with identity and shared references.

Further Reading

- Microsoft Docs: Types in C#: https://learn.microsoft.com/dotnet/csharp/language-reference/builtin-types/built-in-types
- Microsoft Docs: Value Types and Reference Types: https://learn.microsoft.com/dotnet/csharp/programming-guide/types/

C# Basics: Variables, Operators, and Expressions

Variables

Variables are named storage locations in memory that hold data. In C#, you must declare a
0 variable with a specific data type before using it. This helps the compiler allocate the right

amount of memory and enforce type safety.

Key Points:

- Variables must be declared before use.
- The data type determines what kind of data the variable can store.
- Variable names should be descriptive and follow C# naming conventions (camelCase for local variables).

Operators

Operators are symbols that perform operations on variables and values. C# includes several types of operators:

- Arithmetic Operators: For mathematical operations (e.g., +, -, *, /, %)
- Assignment Operators: For assigning values (e.g., =, +=, -=)
- Comparison Operators: For comparing values (e.g., ==, !=, <, >, <=, >=)
- Logical Operators: For logical operations (e.g., &&, ||, !)

Expressions

An expression is a combination of variables, values, and operators that produces a result. For example, a + b is an expression that adds two variables.

Examples

Declarations and arithmetic:

```
int x = 10, y = 3;
int sum = x + y;     // 13
int product = x * y; // 30
int quotient = x / y;  // 3 (integer division)
int remainder = x % y; // 1
```

Comparison and logical:

```
bool isGreater = x > y;              // true
bool bothPositive = (x > 0) && (y > 0); // true
bool eitherLarge = (x >= 10) || (y >= 10); // true
```

0

Precedence and grouping:

```
int result = x + y * 2;   // 10 + 3*2 = 16
int clearer = (x + y) * 2; // 26
```

Best Practices

- Use meaningful variable names.
- Keep expressions simple and readable.
- Use parentheses to clarify complex expressions.

Further Reading

- Microsoft Docs: Variables: https://learn.microsoft.com/dotnet/csharp/programming-guide/variables/
- Microsoft Docs: Operators: https://learn.microsoft.com/dotnet/csharp/language-reference/operators/
- Microsoft Docs: Expressions: https://learn.microsoft.com/dotnet/csharp/language-reference/operators/expressions

---

Type Conversion in C# (Implicit/Explicit, Boxing/Unboxing)

Why Conversion Matters

C# is statically typed, so types must match. Conversions let values move between compatible types with predictable behavior.

Implicit vs Explicit Conversion

- Implicit conversions are safe and lossless (e.g., smaller numeric type to larger). The compiler applies them automatically.
- Explicit conversions require intent because information may be lost or the conversion
0 may fail at runtime.

Numeric Conversions

- Widening (safe): smaller range/precision to larger range/precision.
- Narrowing (risky): larger to smaller; may overflow, truncate, or throw at runtime if checked.

Reference Conversions

- Upcast (derived to base) is safe conceptually.
- Downcast (base to derived) requires runtime type compatibility.

Boxing/Unboxing

- Boxing: wrapping a value type instance as an object to treat it as a reference type.
- Unboxing: extracting the value type from an object; requires the exact original value type.
- Performance note: boxing allocates on the heap and can pressure GC; avoid in hot paths.

Best Practices

- Prefer implicit conversions when they are guaranteed safe.
- Be explicit and intentional with narrowing conversions; validate ranges.
- Minimize boxing by using generics and avoiding APIs that require object.

Examples

Implicit vs explicit and overflow checking:

```
int small = 123;
long bigger = small; // implicit widening

double pi = 3.14;
int truncated = (int)pi; // explicit narrowing => 3

try
{
        checked
        {
                int max = int.MaxValue;
```

0

```
                int overflow = max + 1; // throws OverflowException in
checked context
        }
}
catch (OverflowException)
{
        // handle
}

// Boxing/unboxing
object boxed = small;         // boxing
int unboxed = (int)boxed;     // unboxing
```

Namespaces in C#

Purpose of Namespaces

Namespaces organize types and prevent naming collisions across libraries and projects.

Key Concepts

- Logical grouping: types with related purpose live together.
- Disambiguation: identical type names can coexist in different namespaces.
- Using directives: bring a namespace into scope to shorten type names.
- Aliases: assign a local alias to a type or namespace to avoid ambiguity.

Design Tips

- Mirror folder structure with namespaces for clarity.
- Use company/product root (e.g., Company.Product.Module).
- Avoid deep nesting unless it communicates meaningful boundaries.

Examples

0

Using directives and aliases:

```
using System.Text;                  // bring types into scope
using Col = System.Collections.Generic; // alias

namespace Demo.Project;

public class Example
{
        public string JoinWords(Col.List<string> words)
                => string.Join(' ', words);
}
```

Disambiguation with fully-qualified names:

```
// If two types have the same name in different namespaces
global::System.Uri uri = new("https://example.com");
```

Branching in C# (if/else, switch)

What and Why

Branching lets a program choose different execution paths based on conditions. It’s fundamental to decision-making logic and input validation.

if / else

- Evaluate a boolean condition to choose a path.
- Chain with else if for multiple cases; prefer early returns (guard clauses) for readability.

Example:

```
int score = 78;
if (score < 0 || score > 100)
{
        throw new ArgumentOutOfRangeException(nameof(score));
```

0

```
}
else if (score >= 90)
{
        Console.WriteLine("A");
}
else if (score >= 80)
{
        Console.WriteLine("B");
}
else
{
        Console.WriteLine("C or below");
}
```

switch

- Good for discrete choices based on a single value.
- Pattern matching unlocks matching on types, ranges, and conditions.

Examples:

```
string GradeCategory(int score) => score switch
{
        >= 90 => "Excellent",
        >= 80 => "Good",
        >= 70 => "Fair",
        _ => "Needs Improvement"
};

// Type pattern example
string Describe(object o) => o switch
{
        null => "null",
        string s when s.Length == 0 => "empty string",
        string s => $"string of length {s.Length}",
        int n => $"int {n}",
        _ => o.GetType().Name
};
```

0

Best Practices

- Keep conditions simple and intention-revealing.
- Prefer switch for many discrete cases; avoid long if/else chains.
- Extract complex conditions into well-named helpers for readability and reuse.
- Avoid duplication: compute a value once and reuse it.
- Use guard clauses to fail fast when inputs are invalid.

Looping in C# (for, while, foreach)

What and Why

Loops repeat work over a sequence or until a condition changes. They help process collections, perform retries, and implement state machines.

for / while

- for: use when you control an index and have clear start/stop/step.
- while: use when you loop until a condition becomes false.

Examples:

```
int total = 0;
for (int i = 1; i <= 3; i++)
{
        total += i; // 1+2+3
}

int n = 3;
while (n > 0)
{
        n--; // 3,2,1 -> stop when 0
}
```

foreach

0

- Iterates elements of a collection in sequence order.
- Emphasizes the element rather than index bookkeeping.

```
var items = new[] { "a", "b", "c" };
foreach (var it in items)
{
        Console.WriteLine(it);
}
```

Pitfalls and Tips

- Avoid off-by-one errors by defining inclusive/exclusive bounds explicitly.
- Ensure loop termination; mutate conditions correctly.
- Prefer foreach for readability when indexing isn't needed.
- Use break/continue judiciously; they can simplify control flow but overuse harms clarity.

```
foreach (var word in words)
{
        if (string.IsNullOrWhiteSpace(word)) continue; // skip blanks
        if (word == "STOP") break;                    // early exit
        Console.WriteLine(word);
}
```

Iterators and `yield`

Iterators generate sequence elements on demand with minimal memory and clear code. In C#, you implement iterators with `yield return` and `yield break`, and the compiler builds the underlying state machine for `IEnumerable/IEnumerator`.

When to use

- Stream large or expensive data lazily (avoid loading everything into memory).
- Compose pipelines (filter, map) without intermediate allocations.
- Model infinite or open-ended sequences safely.

0

The iterator contract

- `IEnumerable<T>.GetEnumerator()` returns an `IEnumerator<T>`.
- `IEnumerator<T>` has `bool MoveNext()`, `T Current { get; }`, and `void Reset()` (rarely used), plus `IDisposable`.
- An iterator method that uses `yield` implicitly implements this contract for you.

Basics: `yield return` and `yield break`

```
IEnumerable<int> FirstN(int count)
{
    for (int i = 1; i <= count; i++)
        yield return i; // execution suspends here until next MoveNext()
}

// End a sequence early
IEnumerable<int> OddsUntil(int limit)
{
    for (int i = 1; ; i += 2)
    {
        if (i > limit) yield break;
        yield return i;
    }
}
```

Usage:

```
foreach (var n in FirstN(3))
    Console.WriteLine(n); // 1 2 3

Console.WriteLine(string.Join(", ", OddsUntil(7))); // 1, 3, 5, 7
```

Real-world: lazy file processing

Prefer `File.ReadLines` (lazy) to `ReadAllLines` (eager) for large files.

0

```
IEnumerable<string> ErrorLines(string path)
{
    foreach (var line in File.ReadLines(path)) // streams lines lazily
        if (line.Contains("ERROR"))
            yield return line;
}

// Consumers can bail early without reading the whole file
var firstError = ErrorLines("app.log").FirstOrDefault();
```

Composing iterators

```
IEnumerable<int> Range(int start, int count)
{
    for (int i = 0; i < count; i++)
        yield return start + i;
}

IEnumerable<int> Squares(IEnumerable<int> numbers)
{
    foreach (var n in numbers)
        yield return n * n;
}

var firstFiveSquares = Squares(Range(1, 5)); // 1, 4, 9, 16, 25
```

State, exceptions, and cleanup

- State machine: Local variables are preserved between `yield returns`.
- Exceptions thrown inside the iterator surface at enumeration time (when `MoveNext()` runs).
- Use `try/finally` to guarantee cleanup at the end of enumeration.

```
IEnumerable<string> ReadLinesWithFooter(string path)
{
    using var reader = new StreamReader(path);
    string? line;
```

0

```
    try
    {
        while ((line = reader.ReadLine()) is not null)
            yield return line;
    }
    finally
    {
        yield return "-- EOF --"; // allowed: finally runs on normal or early termination
    }
}
```

Note: In iterators, `using` translates to `try/finally` so the resource is disposed when enumeration completes or is abandoned.

Common pitfalls and tips

- Multiple enumeration repeats work. If you need to iterate multiple times, materialize once: `var cache = source.ToList();`.
- Side effects happen on enumeration, not declaration. Be mindful when passing an `IEnumerable<T>` around.
- Don't capture mutable outer variables you later change; it can lead to confusing results.
- Prefer returning `IEnumerable<T>` over concrete collections when laziness is desired.

Async streams (brief)

For async producers/consumers, use `IAsyncEnumerable<T>` with `await foreach` and `yield return` in async iterator methods.

```
async IAsyncEnumerable<int> Tick(int intervalMs, [EnumeratorCancellation] CancellationToken ct = default)
{
    int i = 0;
    while (!ct.IsCancellationRequested)
    {
        await Task.Delay(intervalMs, ct);
        yield return ++i;
```

0

```
    }
}
```

Common Language Runtime (CLR)

The CLR is the virtual machine that runs .NET code. It loads assemblies, verifies IL, JIT-compiles methods to native code, and manages memory and execution.

Role of CLR

- IL → native via Just-In-Time (JIT) compilation with tiered compilation (fast Tier0 → optimized Tier1).
- Memory management with a generational, concurrent, compacting Garbage Collector.
- Type safety, verification, security boundaries, exception handling.

Key Services

- Garbage Collection: Generations (0/1/2), Large Object Heap (LOH), Server vs Workstation GC, Background GC.
- JIT: Tiered JIT, ReadyToRun (AOT-like precompiled IL), PGO (profile-guided optimization).
- Type System & Metadata: reflection, attributes, runtime type info (RTTI).
- Loading & Isolation: Assemblies, AssemblyLoadContext (plugin isolation), single-file publish.

Practical effects

- Startup vs throughput: tiered JIT improves startup with later optimizations.
- Allocation patterns matter: short-lived objects die young (Gen0) → cheap; avoid LOH fragmentation.
- Exceptions are expensive when thrown; using them for control flow hurts performance.

Interop (brief)

- P/Invoke to call native functions; `DllImport` attribute defines the boundary.

0

```
using System.Runtime.InteropServices;

static class Native
{
        [DllImport("kernel32.dll")]
        public static extern void Sleep(uint dwMilliseconds);
}

Native.Sleep(100);
```

Diagnostics hooks

- ETW/EventPipe (dotnet-trace), dotnet-counters, dotnet-gcdump, PerfView.
- In-process: `GC.GetTotalMemory`, `GC.TryStartNoGCRegion`, `Activity` for tracing.

.NET Framework Class Library (BCL/FCL)

The BCL/FCL is the standard library for .NET: collections, IO, networking, threading, numerics, etc. Learn its surface area to avoid reinventing wheels.

Common namespaces and anchors

- System, System.Collections.Generic (List, Dictionary<TKey,TValue>, HashSet)
- System.Linq (operators for querying in-memory collections)
- System.IO (File, Directory, streams)
- System.Net.Http (HttpClient)
- System.Text.Json (JSON serialization)
- System.Threading / Tasks (Task, CancellationToken)

Handy examples

0

```
// Collections
var counts = new Dictionary<string,int>
(StringComparer.OrdinalIgnoreCase);
foreach (var w in new[] { "a", "b", "A" }) counts[w] =
```

```
counts.GetValueOrDefault(w) + 1;

// IO
File.WriteAllText("demo.txt", "hello");
var text = File.ReadAllText("demo.txt");

// LINQ
var evens = Enumerable.Range(1, 10).Where(n => n % 2 == 0).ToArray();

// JSON
var json = System.Text.Json.JsonSerializer.Serialize(new { Name = "Ada"
});
var obj =
System.Text.Json.JsonSerializer.Deserialize<Dictionary<string,string>>
(json);

// Tasks & cancellation
using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(1));
try { await Task.Delay(5000, cts.Token); }
catch (TaskCanceledException) { /* expected */ }
```

Tips

- Prefer BCL types first; they're well-tested and supported across runtimes.
- Check for `TryXxx` methods to avoid exceptions for common failure paths.

IDE Setup (Visual Studio / VS Code)

VS Code

- Install C# Dev Kit and .NET Runtime extension pack.
- Ensure .NET SDK installed: `dotnet --info`.
- Create a project: `dotnet new console -n Hello` → build/run: `dotnet run`.

0

Visual Studio

- Workloads: ".NET desktop development", "ASP.NET and web development".
- Use Solution Explorer, launch profiles, integrated test runner, and code analyzers.

Project configuration tips

- Nullable references: <Nullable>enable</Nullable> for safer APIs.
- Implicit usings: <ImplicitUsings>enable</ImplicitUsings> reduces boilerplate.
- Treat warnings as errors in CI: <TreatWarningsAsErrors>true</TreatWarningsAsErrors>.
- Add analyzers: StyleCop/IDEs, or enable Microsoft.CodeAnalysis.NetAnalyzers.

CLI essentials

- `dotnet new`, `dotnet add package`, `dotnet build`, `dotnet test`, `dotnet publish`.
- `dotnet watch run` for hot reload during development.

Classes and Objects

Classes model state and behavior; objects are instances with their own state. Prefer small, cohesive classes with clear responsibilities.

Anatomy of a class

```
public class BankAccount
{
        private decimal _balance;              // encapsulated field
        public string Owner { get; }           // init-only via
constructor
        public decimal Balance => _balance;    // read-only property
(expression-bodied)

        public BankAccount(string owner, decimal openingBalance = 0)
        {
```

```
                Owner = owner ?? throw new
ArgumentNullException(nameof(owner));
                if (openingBalance < 0) throw new
ArgumentOutOfRangeException(nameof(openingBalance));
                _balance = openingBalance;
        }

        public void Deposit(decimal amount)
        {
                if (amount <= 0) throw new
ArgumentOutOfRangeException(nameof(amount));
                _balance += amount;
        }

        public bool TryWithdraw(decimal amount)
        {
                if (amount <= 0) return false;
                if (amount > _balance) return false;
                _balance -= amount;
                return true;
        }
}

// Usage
var acct = new BankAccount("Alice", 100m);
acct.Deposit(50m);
Console.WriteLine(acct.Balance); // 150
```

Properties, init-only, and validation

```
public class Person
{
        private int _age;
        public string FirstName { get; init; } = string.Empty; // init-
only at construction
        public string LastName  { get; init; } = string.Empty;
        public int Age
        {
```

0

```
                get => _age;
                set => _age = value >= 0 ? value : throw new
ArgumentOutOfRangeException();
        }
}

var p = new Person { FirstName = "Ada", LastName = "Lovelace", Age = 28
};
```

Indexers and static members

```
public class WordBag
{
        private readonly Dictionary<string,int> _counts =
new(StringComparer.OrdinalIgnoreCase);
        public int this[string word]
        {
                get => _counts.TryGetValue(word, out var c) ? c : 0;
                set => _counts[word] = value;
        }

        public static WordBag FromText(string text)
        {
                var bag = new WordBag();
                foreach (var w in text.Split(' ',
StringSplitOptions.RemoveEmptyEntries))
                        bag[w]++;
                return bag;
        }
}

var bag = WordBag.FromText("to be or not to be");
Console.WriteLine(bag["be"]); // 1
```

Records for immutable data models

0

```
public record Customer(string Id, string Name);

var c1 = new Customer("42", "Dana");
var c2 = c1 with { Name = "Dana S." }; // non-destructive mutation
Console.WriteLine(c1 == c2); // false (value equality)
```

Object initialization and deconstruction

```
public class Point
{
        public int X { get; init; }
        public int Y { get; init; }
        public void Deconstruct(out int x, out int y) { x = X; y = Y; }
}

var pt = new Point { X = 3, Y = 4 };
var (x, y) = pt; // x=3, y=4
```

OOP Principles

Core pillars: Encapsulation, Inheritance, Polymorphism, and Abstraction. Favor composition over deep inheritance chains.

Encapsulation

Hide state, expose behavior with invariants enforced inside the type.

```
public class Thermostat
{
        private double _temperature;
        public double Temperature
        {
                get => _temperature;
                set => _temperature = Math.Clamp(value, 10, 30); // keep
```

0

```
within safe range
        }
}
```

Inheritance (use sparingly)

```
public abstract class Shape { public abstract double Area(); }
public class Rectangle : Shape
{
        public double Width { get; init; }
        public double Height { get; init; }
        public override double Area() => Width * Height;
}
public class Circle : Shape
{
        public double Radius { get; init; }
        public override double Area() => Math.PI * Radius * Radius;
}

Shape s = new Circle { Radius = 2 };
Console.WriteLine(s.Area());
```

Polymorphism

Overriding via virtual/abstract methods; interface-based polymorphism preferred for decoupling.

```
public interface IPrinter { void Print(string message); }
public class ConsolePrinter : IPrinter { public void Print(string m) =>
Console.WriteLine(m); }
public class UpperCasePrinter : IPrinter { public void Print(string m) =>
Console.WriteLine(m.ToUpperInvariant()); }

void Notify(IPrinter printer) => printer.Print("Hello");
```

0

Abstraction

Express intent without committing to details.

```
public interface IRepository<T>
{
        T? Get(string id);
        void Add(T entity);
}
```

Composition over inheritance

```
public class CachedRepository<T> : IRepository<T>
{
        private readonly IRepository<T> _inner;
        private readonly Dictionary<string,T> _cache = new();
        public CachedRepository(IRepository<T> inner) => _inner = inner;

        public T? Get(string id)
        {
                if (_cache.TryGetValue(id, out var v)) return v;
                var e = _inner.Get(id);
                if (e is not null) _cache[id] = e;
                return e;
        }
        public void Add(T entity) => _inner.Add(entity);
}
```

Advanced OOP

Structs vs Classes

- Structs are value types; copied by value, allocated inline when possible.
- Prefer for small, immutable data (e.g., 2–3 fields). Avoid large or mutable structs.

```
public readonly struct Money
{
```

```
        public decimal Amount { get; }
        public string Currency { get; }
        public Money(decimal amount, string currency) { Amount = amount;
Currency = currency; }
        public override string ToString() => $"{Amount} {Currency}";
}
```

Enums & Flags

```
[Flags]
public enum FileAccessRights { None = 0, Read = 1, Write = 2, Execute = 4
}
var rights = FileAccessRights.Read | FileAccessRights.Write;
bool canWrite = rights.HasFlag(FileAccessRights.Write);
```

Nested types

Keep helpers close to usage; avoid overexposure of internals.

```
public class Parser
{
        public sealed class Result { public bool Success { get; init; }
public string? Error { get; init; } }
}
```

Partial types/members

Split large types across files or generate parts via source generators.

```
public partial class UserService { partial void OnCreated(); }
public partial class UserService { partial void OnCreated() { /* hook */
} }
```

Operator overloads (use judiciously)

0

```
public readonly record struct Vector2(double X, double Y)
{
        public static Vector2 operator +(Vector2 a, Vector2 b) => new(a.X
+ b.X, a.Y + b.Y);
}
```

Equality semantics

- Classes default to reference equality; override `Equals/GetHashCode` or use records for value semantics.

```
public record Person(string First, string Last);
var a = new Person("Ada","Lovelace");
var b = new Person("Ada","Lovelace");
Console.WriteLine(a == b); // true (value-based)
```

Best practices

- Favor immutability where practical.
- Keep constructors simple; use factories/builders if setup is complex.
- Keep inheritance shallow; prefer interfaces + composition.

Built-in Collections

Choose the right structure for performance and clarity. Know the complexity and common pitfalls.

Core types and when to use

- Array (T[]): fixed size, contiguous memory, fastest indexing.
- List: dynamic array, amortized O(1) append, O(1) index.
- Dictionary<TKey,TValue>: hash map, O(1) average lookup/insert.
- HashSet: uniqueness set, O(1) average contains/add.
- Queue, Stack: FIFO/LIFO with O(1) enqueue/dequeue/push/pop.

0

- LinkedList: O(1) insert/remove with node, O(n) indexing; niche use.
- Concurrent collections: thread-safe data structures for multi-producer/consumer.

Idiomatic examples

```
// List
var nums = new List<int> { 1, 2, 3 };
nums.Add(4);
nums.RemoveAll(n => n % 2 == 0); // 1,3

// Dictionary
var counts = new Dictionary<string,int>
(StringComparer.OrdinalIgnoreCase);
foreach (var w in new[] { "A", "b", "a" })
        counts[w] = counts.GetValueOrDefault(w) + 1;

// HashSet
var unique = new HashSet<int> { 1, 2, 2, 3 }; // 1,2,3

// Queue/Stack
var q = new Queue<string>(); q.Enqueue("first"); q.Enqueue("second"); var
head = q.Dequeue();
var s = new Stack<string>(); s.Push("x"); s.Push("y"); var top = s.Pop();
```

Concurrent collections

```
var bag = new System.Collections.Concurrent.ConcurrentBag<int>();
Parallel.For(0, 1000, bag.Add);
int count = bag.Count; // thread-safe aggregation pattern differs

var queue = new System.Collections.Concurrent.BlockingCollection<int>();
var prod = Task.Run(() => { for (int i = 0; i < 10; i++) queue.Add(i);
queue.CompleteAdding(); });
var cons = Task.Run(() => { foreach (var item in
queue.GetConsumingEnumerable()) Console.WriteLine(item); });
await Task.WhenAll(prod, cons);
```

0

When to use which:

- Use ConcurrentDictionary when multiple threads update shared counters/state per key.
- Use BlockingCollection for producer/consumer pipelines with backpressure.
- Prefer immutable snapshots (e.g., ImmutableArray) for many-readers/few-writers patterns.

Complexity cheatsheet (typical)

- List: index O(1), append amortized O(1), remove by value O(n).
- Dictionary/HashSet: add/contains O(1) average; O(n) worst-case.
- Queue/Stack: O(1) enqueue/dequeue/push/pop.

Tips

- Prefer `TryGetValue/GetValueOrDefault` to avoid exceptions on missing keys.
- Use `StringComparer.OrdinalIgnoreCase` when keys are case-insensitive.
- Avoid repeated `List<T>.Remove(item)` in a loop; filter with `Where/RemoveAll`.

Further reading

- https://learn.microsoft.com/dotnet/standard/collections/

Custom Collections

Implementing custom collections lets you enforce invariants and expose efficient operations. Prefer composition and interfaces.

Implementing IEnumerable

```
public sealed class EvenNumbers : IEnumerable<int>
{
        private readonly int _limit;
        public EvenNumbers(int limit) => _limit = limit;
        public IEnumerator<int> GetEnumerator() => Iterator();
```

0

```
        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() => GetEnumerator();
        private IEnumerator<int> Iterator()
        {
                for (int i = 0; i <= _limit; i += 2)
                        yield return i;
        }
}
```

Implementing IList (sketch)

```
public class BoundedList<T> : IList<T>
{
        private readonly List<T> _inner = new();
        public int Capacity { get; }
        public BoundedList(int capacity) => Capacity = capacity;
        public T this[int index] { get => _inner[index]; set =>
_inner[index] = value; }
        public int Count => _inner.Count;
        public bool IsReadOnly => false;
        public void Add(T item) { if (Count >= Capacity) throw new
InvalidOperationException("Full"); _inner.Add(item); }
        public void Clear() => _inner.Clear();
        public bool Contains(T item) => _inner.Contains(item);
        public void CopyTo(T[] array, int arrayIndex) =>
_inner.CopyTo(array, arrayIndex);
        public IEnumerator<T> GetEnumerator() => _inner.GetEnumerator();
        public int IndexOf(T item) => _inner.IndexOf(item);
        public void Insert(int index, T item) { if (Count >= Capacity)
throw new InvalidOperationException("Full"); _inner.Insert(index, item);
}
        public bool Remove(T item) => _inner.Remove(item);
        public void RemoveAt(int index) => _inner.RemoveAt(index);
        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() => _inner.GetEnumerator();
}
```

0

Exception Handling

Exceptions represent exceptional, non-expected paths. Use them to signal failure, not for normal branching.

Basics: try/catch/finally

```
try
{
        using var stream = File.OpenRead("config.json");
        // work with stream
}
catch (FileNotFoundException ex)
{
        Console.Error.WriteLine($"Missing config: {ex.FileName}");
}
catch (IOException ex) when (ex.HResult == -2147024864) // example of
filter (file in use)
{
        Console.Error.WriteLine("File is locked.");
}
catch (Exception ex)
{
        Console.Error.WriteLine($"Unexpected: {ex}");
        throw; // rethrow preserving stack trace
}
finally
{
        // cleanup that must always run
}
```

Best practices

- Catch narrowly; let higher layers handle what they own.
- Use exception filters (`catch (X ex) when (...)`) to avoid partial state changes.
- Don’t swallow exceptions silently; log with context.
- Prefer `TryXxx` patterns (e.g., `int.TryParse`) when failure is expected.

0

Creating error context

```
try
{
        ProcessOrder(orderId);
}
catch (OrderStorageException ex)
{
        throw new OrderProcessingException($"Could not process order
{orderId}", ex);
}
```

Custom Exceptions

Define custom exceptions to convey domain-specific failures and enable precise handling.

Template

```
[Serializable]
public class OrderProcessingException : Exception
{
        public string? OrderId { get; }
        public OrderProcessingException() { }
        public OrderProcessingException(string message) : base(message) {
}
        public OrderProcessingException(string message, Exception inner)
: base(message, inner) { }
        public OrderProcessingException(string message, string orderId) :
base(message) => OrderId = orderId;
        protected
OrderProcessingException(System.Runtime.Serialization.SerializationInfo
info,

System.Runtime.Serialization.StreamingContext context)
```

0

```
            : base(info, context) { }
}
```

Tips

- Name them clearly; include meaningful properties (like identifiers).
- Preserve inner exceptions; they're essential for root-cause analysis.
- Avoid throwing exceptions for control flow; use `TryXxx` when failure is common.

Debugging Techniques

Debugging is about fast feedback and narrowing hypotheses.

Core tools

- Breakpoints (conditions, hit counts), data tips, watch/locals, call stack, step-into/out/over.
- Edit and Continue, exception settings (break on thrown/unhandled).

Logging

```
using Microsoft.Extensions.Logging;

using var loggerFactory = LoggerFactory.Create(b =>
b.AddSimpleConsole().SetMinimumLevel(LogLevel.Debug));
var logger = loggerFactory.CreateLogger("Demo");
logger.LogInformation("Starting module {Module}", "X");
```

Tactics

- Reproduce deterministically; reduce the surface (disable concurrency, mock IO).
- Bisect changes (git); add asserts for invariants.
- Capture context: inputs, environment, timing, correlation IDs.

0

Performance debugging

- dotnet-trace/dotnet-counters; sampling profilers; memory dumps (dotnet-gcdump).

Delegates and Events

Delegates are type-safe function references; events build a publish/subscribe layer on top.

Delegates and built-ins

```
// Custom delegate type
public delegate int BinaryOp(int a, int b);
int Add(int x, int y) => x + y;
BinaryOp op = Add;
int r = op(2, 3); // 5

// Built-ins
Action<string> log = Console.WriteLine;     // no return
Func<int,int,int> mul = (a,b) => a * b;     // returns int
Predicate<int> isEven = n => n % 2 == 0;    // bool-returning
Func<T,bool>
```

Lambdas and closures

```
int factor = 10;                    // captured variable
Func<int,int> times = n => n * factor;
factor = 20;                        // closure observes latest value
Console.WriteLine(times(2));       // 40
```

Multicast delegates

```
Action pipeline = () => Console.Write("A");
pipeline += () => Console.Write("B");
```

0

```
pipeline(); // prints AB
```

Events (EventHandler pattern)

```
public class Counter
{
        public event EventHandler<int>? ThresholdReached; // payload via
generic arg
        private int _count;
        public void Increment()
        {
                _count++;
                if (_count % 5 == 0)
                        ThresholdReached?.Invoke(this, _count); // raise
safely with null-conditional
        }
}

var c = new Counter();
c.ThresholdReached += (s, value) => Console.WriteLine($"Hit {value}");
for (int i=0;i<10;i++) c.Increment();
```

Custom event accessors (advanced)

```
private EventHandler? _handlers;
public event EventHandler Something
{
        add { _handlers = (EventHandler?)Delegate.Combine(_handlers,
value); }
        remove { _handlers = (EventHandler?)Delegate.Remove(_handlers,
value); }
}
```

Tips

- Prefer Action/Func over custom delegate types unless naming adds clarity.

0

- Be careful with closures in loops; capture the loop variable into a local.
- Unsubscribe from long-lived events to avoid memory leaks.

LINQ

LINQ provides declarative querying for objects, XML, databases, and more.

Two styles

```
// Query syntax
var q = from n in Enumerable.Range(1, 10)
                where n % 2 == 0
                select n * n;

// Method syntax
var m = Enumerable.Range(1,10).Where(n => n % 2 == 0).Select(n => n * n);
```

Core operators

- Filtering: Where
- Projection: Select, SelectMany
- Sorting: OrderBy/ThenBy
- Grouping: GroupBy
- Joining: Join, GroupJoin
- Set ops: Distinct, Union, Intersect, Except
- Aggregates: Count, Sum, Min/Max, Average, Aggregate

```
var people = new[] {
        new { Name = "Ann", City = "NY", Age = 30 },
        new { Name = "Bob", City = "SF", Age = 25 },
        new { Name = "Cat", City = "NY", Age = 40 },
};

var byCity = people.GroupBy(p => p.City)
                                   .Select(g => new { City = g.Key,
```

0

```
AvgAge = g.Average(p => p.Age) });

var orders = new[] { new { Id = 1, Customer = "Ann" } };
var customers = new[] { new { Name = "Ann" }, new { Name = "Bob" } };
var join = from o in orders
                   join c in customers on o.Customer equals c.Name
                   select new { o.Id, o.Customer };
```

Deferred vs immediate execution

- Deferred: Where/Select build a pipeline evaluated on enumeration.
- Immediate: ToList/ToArray/Count materialize or compute immediately.

```
var source = new List<int> { 1, 2 };
var seq = source.Select(n => n * 10); // deferred
source.Add(3);
var arr = seq.ToArray(); // 10, 20, 30
```

IEnumerable vs IQueryable

- IEnumerable: in-memory; operators run as .NET delegates.
- IQueryable: expression trees; provider can translate to SQL or other backends. Beware of unsupported methods.

Tips

- Push filters early (Where) and project only what you need (Select) to reduce work.
- Avoid multiple enumeration if source is expensive; materialize once when needed.

Practice

- Given orders with a CustomerId, output the top 3 orders by total per customer.
- Inner join vs group join: produce both and explain the shape differences.
- Flatten nested collections (customers -> orders -> lines) and compute totals with SelectMany.

0

Asynchronous Programming

Use async/await to free threads while work is pending (IO), improving scalability and responsiveness.

async/await basics

```
async Task<string> DownloadAsync(HttpClient http, string url)
{
        var resp = await http.GetAsync(url); // awaits without blocking
        resp.EnsureSuccessStatusCode();
        return await resp.Content.ReadAsStringAsync();
}
```

Cancellation and timeouts

```
using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(2));
try
{
        await Task.Delay(5000, cts.Token);
}
catch (OperationCanceledException)
{
        // cancelled
}
```

Error handling

```
try { await SomeAsync(); }
catch (HttpRequestException ex) { /* network failure */ }
```

ConfigureAwait

0

In libraries, prefer `await task.ConfigureAwait(false)` to avoid capturing context. In apps (UI), default capture is usually fine.

Parallelism

```
// CPU-bound parallel loop (data parallelism)
Parallel.ForEach(data, item => Process(item));

// Fire multiple IO tasks concurrently and await all
var tasks = urls.Select(http.GetStringAsync);
var pages = await Task.WhenAll(tasks);
```

Tips

- Don’t block on async (no .Result/.Wait()); make your call chain async.
- Use `ValueTask` for high-throughput hot paths when appropriate.

Practice

- Wrap an external API call with timeout and cancellation, surfacing a custom exception on failure.
- Convert a synchronous file processing loop to async and ensure max 4 concurrent operations.
- Explain ConfigureAwait(false) and where it’s appropriate; demonstrate a context-deadlock caused by .Result.

ADO.NET

Low-level data access with explicit connections, commands, and readers. Great for tight control and performance.

Connected: commands and readers

0

```
using var conn = new Microsoft.Data.Sqlite.SqliteConnection("Data
Source=:memory:");
await conn.OpenAsync();
using var cmd = conn.CreateCommand();
cmd.CommandText = "CREATE TABLE T(Id INTEGER PRIMARY KEY, Name TEXT);
INSERT INTO T(Name) VALUES ('Ada'); SELECT Id, Name FROM T;";
using var reader = await cmd.ExecuteReaderAsync();
while (await reader.ReadAsync())
        Console.WriteLine($"{reader.GetInt32(0)} {reader.GetString(1)}");
```

Disconnected: DataTable

```
var table = new System.Data.DataTable();
using (var cmd = conn.CreateCommand())
{
        cmd.CommandText = "SELECT 1 AS N UNION ALL SELECT 2";
        using var reader = await cmd.ExecuteReaderAsync();
        table.Load(reader); // Fast materialization without DataAdapter
}
```

Tips:

- Track RowState (Added/Modified/Deleted) to know what to persist.
- Prefer `DataTable.Load(IDataReader)` for simple reads.
- Keep ADO.NET for surgical control and batching; use EF/Dapper when object mapping productivity is needed.

Transactions

```
using var tx = await conn.BeginTransactionAsync();
try
{
        using var c1 = conn.CreateCommand(); c1.Transaction = tx;
c1.CommandText = "INSERT INTO T(Name) VALUES ('Babbage')"; await
c1.ExecuteNonQueryAsync();
        using var c2 = conn.CreateCommand(); c2.Transaction = tx;
c2.CommandText = "INSERT INTO T(Name) VALUES ('Turing')"; await
```

0

```
c2.ExecuteNonQueryAsync();
        await tx.CommitAsync();
}
catch
{
        await tx.RollbackAsync();
        throw;
}
```

Further reading

- https://learn.microsoft.com/dotnet/framework/data/adonet/ado-net-overview

---

Entity Framework Core

ORM for .NET with LINQ queries and change tracking.

Model & DbContext

```
public class Blog { public int Id { get; set; } public string Title {
get; set; } = ""; public List<Post> Posts { get; set; } = new(); }
public class Post { public int Id { get; set; } public string Content {
get; set; } = ""; public int BlogId { get; set; } public Blog? Blog {
get; set; } }

public class AppDb : DbContext
{
        public DbSet<Blog> Blogs => Set<Blog>();
        public DbSet<Post> Posts => Set<Post>();
        protected override void OnConfiguring(DbContextOptionsBuilder b)
=> b.UseSqlite("Data Source=app.db");
        protected override void OnModelCreating(ModelBuilder mb) =>
mb.Entity<Post>().HasIndex(p => p.BlogId);
}
```

0

Queries and tracking

```
using var db = new AppDb();
db.Database.EnsureCreated();
db.Blogs.Add(new Blog { Title = "Hello" });
db.SaveChanges();

var blogs = await db.Blogs.AsNoTracking().Where(b =>
b.Title.Contains("H")).ToListAsync();
```

Migrations (concept)

- Add: dotnet ef migrations add Initial
- Update DB: dotnet ef database update
- Track schema changes over time; commit migration files.

Tips

- Scope DbContext per unit of work (e.g., per web request).
- Use AsNoTracking for read-only queries; include navigation properties with `.Include` when needed.

Practice

- Add a unique index to Blog.Title using Fluent API and verify the constraint.
- Demonstrate tracking vs AsNoTracking and explain memory/perf impact in a list view.
- Implement a one-to-many with cascade delete and write a test to verify.

File I/O

Streams

```
await using var fs = new FileStream("data.bin", FileMode.Create,
FileAccess.Write, FileShare.None, 8192, useAsync: true);
```

0

```
var bytes = Encoding.UTF8.GetBytes("hello");
await fs.WriteAsync(bytes);
```

Text convenience

```
File.WriteAllText("greet.txt", "hi");
var text = File.ReadAllText("greet.txt");
```

JSON serialization

```
record Person(string Name, int Age);
var json = System.Text.Json.JsonSerializer.Serialize(new Person("Ada",
28));
var p = System.Text.Json.JsonSerializer.Deserialize<Person>(json);
```

XML serialization

```
var xmlSer = new System.Xml.Serialization.XmlSerializer(typeof(Person));
await using var xfs = File.Create("person.xml");
xmlSer.Serialize(xfs, new Person("Ada", 28));
```

Tips

- Prefer async IO for scalability in servers; sync is often fine for small local work.
- Use File.ReadLines (lazy) over ReadAllLines (eager) for large files.

WPF: XAML Basics

Layouts

0

```
<Grid>
        <Grid.RowDefinitions>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <StackPanel Orientation="Horizontal" Margin="8">
                <TextBlock Text="Name:" Margin="0,0,8,0"/>
                <TextBox Width="200" Text="{Binding Name,
UpdateSourceTrigger=PropertyChanged}"/>
        </StackPanel>
        <ListBox Grid.Row="1" ItemsSource="{Binding Items}"/>
        </Grid>
```

Data Binding with INotifyPropertyChanged

```
public class MainViewModel : INotifyPropertyChanged
{
                private string _name = string.Empty;
                public string Name { get => _name; set { if
(_name!=value){ _name=value; OnPropertyChanged(); } } }
                public ObservableCollection<string> Items { get; } =
new();
                public event PropertyChangedEventHandler?
PropertyChanged;
                void OnPropertyChanged([CallerMemberName] string? n=null)
=> PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(n));
}
```

Commands (basic)

```
public class RelayCommand : ICommand
{
                private readonly Action _exec; private readonly
Func<bool>? _can;
                public RelayCommand(Action exec, Func<bool>? can=null)
{_exec=exec;_can=can;}
0               public event EventHandler? CanExecuteChanged;
```

```
                public bool CanExecute(object? p)=>_can?.Invoke()??true;
                public void Execute(object? p)=>_exec();
                public void
RaiseCanExecuteChanged()=>CanExecuteChanged?.Invoke(this,
EventArgs.Empty);
}
```

Binding modes and validation

- Modes: OneTime, OneWay, TwoWay (default for TextBox.Text), OneWayToSource.
- Validation: IDataErrorInfo/INotifyDataErrorInfo; ValidationRules on bindings.

Practice

- Bind a Slider to a numeric property (TwoWay) and display its value.
- Add validation to disallow empty names and show a red adornment.

WPF: Advanced

Styles and Templates

```
<Window.Resources>
        <Style TargetType="Button">
                <Setter Property="Margin" Value="4"/>
        </Style>
        <DataTemplate DataType="{x:Type vm:Person}">
                <StackPanel Orientation="Horizontal">
                        <TextBlock Text="{Binding First}"/>
                        <TextBlock Text=" "/>
                        <TextBlock Text="{Binding Last}"/>
                </StackPanel>
        </DataTemplate>
 </Window.Resources>
```

0

Commands and MVVM

```
public class MainViewModel
{
                public ObservableCollection<Person> People { get; } =
new();
                public ICommand AddPerson { get; }
                public MainViewModel(){ AddPerson = new RelayCommand(()
=> People.Add(new Person("Ada","Lovelace"))); }
}
```

Binding diagnostics

- Use PresentationTraceSources for binding debug.
- Enable exceptions on binding failures in dev.

ASP.NET Core Fundamentals

Middleware pipeline

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.Use(async (ctx, next) => { Console.WriteLine($"{ctx.Request.Path}");
await next(); });
app.MapGet("/hello", () => "world");
app.Run();
```

Razor Pages vs MVC

- Razor Pages: page-focused, good for simple apps.
- MVC: controllers/views, better for larger apps and separation concerns.

Minimal APIs

0

```
app.MapPost("/sum", (int a, int b) => Results.Ok(new { sum = a + b }));
```

Web API essentials

- Model binding, validation attributes, filters, content negotiation (JSON by default).

Blazor

Component basics

```
@page "/counter"
<h3>Counter</h3>
<p>Current count: @count</p>
<button class="btn btn-primary" @onclick="Increment">Click me</button>
@code { int count; void Increment() => count++; }
```

Parameters and cascading values

```
<MyCard Title="Hello">Content</MyCard>

@code {
        [Parameter] public string Title { get; set; } = string.Empty;
}
```

Dependency injection

```
@inject HttpClient Http
@code {
        protected override async Task OnInitializedAsync() { var data =
await Http.GetStringAsync("/api"); }
}
```

0

Hosting models

- Server: thin client, low download, requires persistent connection.
- WebAssembly: runs in browser, offline capable, larger download.

Web Security

Authentication

- Cookies (server-rendered sites) vs JWT (APIs/SPAs). External providers via OAuth/OIDC.

```
builder.Services.AddAuthentication("Bearer").AddJwtBearer();
```

Authorization

Security checklist (practical)

- Enforce HTTPS; add HSTS in production.
- Validate and encode all inputs/outputs to prevent XSS/SQLi.
- Use ASP.NET Core Data Protection for key management.
- Store secrets outside source control (User Secrets/Azure Key Vault).
- Implement proper CORS policy (allow only known origins, methods, headers).
- Add rate limiting for public endpoints.
- Log auth failures and suspicious activities; monitor with alerts.
- Roles: [Authorize(Roles = "Admin")]
- Policies: configure requirements centrally.

```
builder.Services.AddAuthorization(o => o.AddPolicy("AdultOnly", p =>
p.RequireClaim("age", "18+")));
app.MapGet("/secure", [Authorize(Policy="AdultOnly")] () => "ok");
```

HTTPS & CORS

0

```
app.UseHttpsRedirection();
app.UseCors(p =>
p.WithOrigins("https://example.com").AllowAnyHeader().AllowAnyMethod());
```

Razor Pages vs MVC in ASP.NET Core

Understand when to choose Razor Pages or MVC:

- Razor Pages: Page-focused, minimal ceremony, great for simple CRUD and forms. Files live side-by-side (.cshtml + PageModel).
- MVC: Controller-centric, great for larger apps, strong separation of concerns, filters, and complex routing.

Key differences:

- Handler methods (OnGet/OnPost) in Pages vs Controller actions.
- Routing conventions: folder-based for Pages vs attribute/conventional for MVC.
- View models: both support, MVC often uses dedicated DTOs and services.

When to pick:

- Small to medium apps, internal tools → Razor Pages.
- Complex APIs, multiple controllers, rich filters → MVC.

Tip: You can mix both in one app.

Xamarin.Forms

Note: .NET MAUI is the modern successor; concepts are similar.

XAML Layouts

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
```

```
x:Class="Sample.MainPage">
        <StackLayout Padding="20">
                <Label Text="Hello"/>
                <Entry Text="{Binding Name}"/>
                <Button Text="Click" Command="{Binding SayHello}"/>
        </StackLayout>
 </ContentPage>
```

Navigation

```
await Navigation.PushAsync(new DetailsPage());
```

MVVM

- Bind View to ViewModel properties/commands via INotifyPropertyChanged and ICommand.

Mobile Features

Local Storage (SQLite.NET)

```
using SQLite;
public class Person { [PrimaryKey, AutoIncrement] public int Id { get;
set; } public string Name { get; set; } = ""; }
var db = new SQLiteAsyncConnection(dbPath);
await db.CreateTableAsync<Person>();
await db.InsertAsync(new Person { Name = "Ada" });
```

Platform-specific code

0

```
public interface IDeviceInfo { string GetModel(); }
// Implement per platform and register with DependencyService or via MAUI
handlers.
```

OAuth 2.0 / OIDC

- Use the system browser; follow the authorization code flow with PKCE.
- Store tokens securely (Keychain/Keystore); refresh tokens carefully.

.NET MAUI Intro

Modern cross-platform UI framework (Windows, macOS, iOS, Android) succeeding Xamarin.Forms.

Concepts:

- Single project targeting multiple platforms
- XAML UI with MVVM
- Handlers (replacing renderers)
- Essentials (device APIs)

Quick start steps:

1. Install .NET SDK with MAUI workload
2. Create a new MAUI app
3. Run on Windows or Android emulator

Migration notes from Xamarin.Forms:

- Namespaces and APIs updated
- Renderers → Handlers
- Shell navigation remains, improved

When to choose MAUI vs Xamarin:

- New apps: MAUI.
- Existing Xamarin.Forms: plan for migration.

0

Cloud Deployment

Azure App Service (typical flow)

- Publish from CLI: `dotnet publish -c Release` then deploy via Azure CLI or VS.
- Configure app settings/environment variables in App Service (Key Vault for secrets).
- Enable logging and Application Insights.

Docker containers

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
WORKDIR /app
EXPOSE 8080
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY . .
RUN dotnet publish -c Release -o /out
FROM base AS final
WORKDIR /app
COPY --from=build /out .
ENTRYPOINT ["dotnet", "WebApi.dll"]
```

Configuration & secrets

- Use appsettings..json + environment variables; never commit secrets.
- For cloud, prefer managed secret stores (Azure Key Vault, AWS Secrets Manager).

Scaling & health

- Health checks endpoint; autoscaling rules; rolling deployments/slots.

CI/CD Pipelines

0 Automate build, test, and deploy on every change.

GitHub Actions (example)

```
name: build
on: [push, pull_request]
jobs:
        build:
                runs-on: windows-latest
                steps:
                        - uses: actions/checkout@v4
                        - uses: actions/setup-dotnet@v4
                              with: { dotnet-version: '8.0.x' }
                        - run: dotnet restore
                        - run: dotnet build --no-restore -c Release
                        - run: dotnet test --no-build -c Release
```

Practices

- Build/test on every push and PR; enforce quality gates.
- Cache dependencies where possible for speed.
- Version artifacts and publish build outputs (e.g., to GitHub Releases).
- Use environments and approvals for production.

Docker Containers for .NET Apps

This guide shows how to containerize your .NET applications and run them locally. It pairs with the `examples/WebApi` project.

Why containers

- Consistent runtime across machines
- Fast deploys and easy rollbacks
- Great fit for CI/CD and cloud platforms

Minimal Dockerfile (ASP.NET Core)

0

We include a ready-to-use Dockerfile in `examples/WebApi` targeting .NET 8.

Key points:

- Multi-stage build (restore/build/publish runtime image)
- Non-root user for runtime (where supported)
- Expose port 8080 inside the container

Build and run

1. Build image
   - Image name: `learning-webapi:dev`
2. Run container
   - Map host port 8080 to container 8080
   - Hit http://localhost:8080/swagger

Troubleshooting:

- If the port is in use, change host mapping `-p 8081:8080` and browse 8081.
- Ensure HTTPS is disabled or dev certs are handled inside container; our sample uses HTTP for simplicity.

Next steps

- Push to a registry (Docker Hub, GHCR, ACR)
- Deploy to Azure Web App for Containers or Kubernetes

Exam Cram: C#/.NET Quick Reference

Use this as your last-minute refresher. Practice from the section prompts in each chapter; this page is for recall.

Core C#

- Value vs reference: structs/enums vs classes/arrays/strings (string is reference/immutable). Passing ref type copies the reference.

0

- Conversions: implicit (safe) vs explicit (cast); checked for overflow; boxing/unboxing for value types.
- Flow: if/else, switch (patterns, when guards), loops (for/while/foreach), break/continue.
- Iterators: `yield return` (lazy), `yield break` (stop). Side effects occur on enumeration.

OOP essentials

- Encapsulation: hide fields; validate in properties; keep invariants.
- Inheritance: `virtual/override/abstract/sealed`; prefer composition for reuse.
- Polymorphism: interfaces or virtual methods; favor interface-first design.
- Records: value semantics and with-expressions; great for immutable DTOs.

Collections: pick fast

- List: ordered, O(1) index, appends amortized O(1).
- Dictionary<TKey,TValue>: O(1) avg lookup; use StringComparer for string keys.
- HashSet: fast uniqueness membership.
- Queue/Stack: FIFO/LIFO O(1) ops; BlockingCollection/ConcurrentBag for threads.
- Avoid repeated List.Remove in loops; prefer RemoveAll/filtering.

Exceptions

- Use exceptions for exceptional paths; not control flow.
- Pattern: try → specific catch → generic catch (log) → finally. Filters: `catch (X ex) when (cond)`.
- Rethrow with `throw;` to preserve stack; prefer `TryXxx` for expected failures.
- Custom exception: serializable, useful properties, preserve inner.

Delegates & events

- Delegates: `Action`, `Func`, `Predicate` cover most needs. Lambdas can capture variables (closures).
- Events: `event EventHandler<T>`; raise with null-conditional; unsubscribe to avoid leaks.

LINQ map

0

- Filter: Where; Project: Select/SelectMany; Sort: OrderBy/ThenBy; Group: GroupBy; Join: Join/GroupJoin; Sets: Distinct/Union/Intersect/Except; Aggregates: Count/Sum/Average/Aggregate.
- Deferred vs immediate: pipelines run on enumeration; materialize with ToList/ToArray when needed.
- IEnumerable vs IQueryable: in-memory vs provider-translated; avoid client-only methods in IQueryable.

Async/await

- Don’t block (no .Result/.Wait); async all the way. Use `Task.WhenAll` for parallel async IO.
- Cancellation: pass CancellationToken; catch OperationCanceledException. Timeouts via CTS.
- Libraries: `ConfigureAwait(false)`; UI apps usually capture context.
- Parallel: CPU-bound → Parallel.ForEach/PLINQ; IO-bound → async + WhenAll.

ADO.NET vs EF Core

- ADO.NET: explicit SqlConnection/Command/Reader; optimal control and perf.
- EF Core: LINQ + change tracking; faster dev, migrations, relationships.
- Transactions: `BeginTransaction` + commit/rollback. Parameters prevent SQL injection.
- EF tips: scope DbContext per unit of work; `AsNoTracking` for read-only; `Include` for navs; migrations: `add` then `update`.

File I/O

- Use async IO on servers; `File.ReadLines` for lazy large files. JSON with System.Text.Json; XML with XmlSerializer.

WPF

- Binding: INotifyPropertyChanged; modes (OneWay, TwoWay). Commands (ICommand) decouple UI.
- Validation: IDataErrorInfo/INotifyDataErrorInfo or ValidationRules on bindings.

ASP.NET Core

0

- Pipeline order: UseRouting → UseAuthentication → UseAuthorization → Map endpoints.
- Minimal API shape: `app.MapGet("/path", (deps, ...) => Results.Ok(...));`
- Model binding, validation attributes, content negotiation (JSON default).

Blazor

- Server vs WASM: latency/connection vs offline/native-like; same component model.
- @inject DI for services; parameters via `[Parameter]`.

Security

- Cookies (server pages) vs JWT (APIs/SPAs). HTTPS always; strict CORS.
- Roles: `[Authorize(Roles="Admin")]`; Policies: central requirements; claims-based.

CLR/BCL

- GC: Gen0/1/2, LOH; allocations cheap when short-lived. Exceptions are costly when thrown.
- JIT: tiered compilation, ReadyToRun; diagnostics via dotnet-trace/counters.
- Prefer BCL types first (collections, IO, HttpClient, JsonSerializer).

DevOps

- CI: restore/build/test on push/PR. Cache deps. Fail fast on warnings.
- Docker: multi-stage build; environment via variables; health checks.
- Cloud: config from env/Key Vault; enable logs and health probes; use slots for safe deploys.

Last-minute checks

- Nullable enabled; guard public APIs.
- Dispose IDisposables (`using`/`await using`).
- Avoid multiple enumeration of expensive sources.
- Validate user input; parameterize SQL; never log secrets.

0