DevSheets

C#

Core syntax for modern .NET.

01 Start a console app

```
Console.WriteLine("Hello, C#!");
string name = "Maya";
Console.WriteLine($"Hi, {name}");
```

Create a project with dotnet new console -n Hello, then cd Hello and dotnet run. Put statements in Program.cs; no Main wrapper is needed.

Examples target C# 10+ with a modern .NET SDK, implicit usings, and nullable checks enabled. Type declarations go after top-level statements.

02 Variables & types

```
int count = 3;
double ratio = 0.5;
decimal price = 19.99m;
bool ready = true;
char grade = 'A';
var name = "Maya";
const int Max = 10;
```

var infers a static type; it is not dynamic. Use decimal for base-10 decimal calculations. const values are fixed at compile time.

03 Operators & parsing

```
int whole = 7 / 2;      // 3
double part = 7 / 2.0;  // 3.5
bool ok = int.TryParse("25", out int age);
string text = age.ToString();
```

Use + - * / %, == != < <= > >=, and && || !. Integer division truncates toward zero. TryParse returns false on failure instead of throwing.

04 Strings

```
string name = " Maya ".Trim();
int size = name.Length;   // 4
name.ToUpperInvariant(); // "MAYA"
name.Contains("ay");     // true
name.Substring(0, 2);    // "Ma"
bool same = name == "Maya";
```

Strings are immutable; methods return new values. == compares string contents. Use $"Hi, {name}" for interpolation and @"C:\temp" for verbatim text.

05 Conditions & patterns

```
int score = 82;
string result = score >= 60 ? "OK" : "Try";
string grade = score switch {
  >= 90 => "A",
  >= 80 => "B",
  _ => "Keep practicing"
};
```

Use if (condition) { ... } else { ... } for branching. A switch expression returns a value; _ provides a fallback pattern.

06 Loops

```
for (int i = 0; i < 3; i++) {
  Console.WriteLine(i);
}
int count = 3;
while (count > 0) { count--; }
```

foreach iterates a collection. break exits the nearest loop; continue moves to its next iteration. do/while runs its body at least once.

07 Arrays & lists

```
int[] scores = { 80, 95, 72 };
int size = scores.Length;
var names = new List<string>();
names.Add("Maya");
string first = names[0];
```

Arrays have fixed length; List<T> can grow. Both use zero-based indices. Lists use Count. foreach (var name in names) visits each element.

08 Methods & tuples

```
static int Add(int a, int b) => a + b;
static (int Min, int Max) Bounds() {
  return (1, 10);
}
var (low, high) = Bounds();
int total = Add(2, 3);
```

Methods declare parameter and return types; void means no result. Here they are top-level local functions. Tuples group a few related values.

09 Nullable values

```
string? name = null;
string label = name ?? "Guest";
int length = name?.Length ?? 0;
int? age = null;
if (age is int years) {
  Console.WriteLine(years);
}
```

? permits null in this context; ?. accesses only when non-null; ?? supplies a fallback. Nullable reference checks are compiler analysis, not runtime enforcement.

C#

Objects and useful .NET APIs.

10 Classes & properties

```
class User {
  public string Name { get; set; }
  public User(string name) {
    Name = name;
  }
}
```

Create an instance with new User("Maya"). Properties control access to data. public is accessible to callers; private stays inside the class. static belongs to the type.

11 Interfaces & inheritance

```
interface IPrintable {
  string Print();
}
class Note : IPrintable {
  public string Print() => "Hi";
}
```

A class can implement multiple interfaces and inherit one base class. Use virtual on a base method and override in the derived class to customize behavior.

12 Records & enums

```
record Point(int X, int Y);
enum Status { Todo, Done }
// Use these in your statements:
// var p = new Point(2, 3);
// var q = p with { X = 5 };
```

Records provide value-based equality and concise data models. with creates a shallow copy with selected changes. Referenced objects may remain mutable. Enums define named choices.

13 Dictionaries & sets

```
var ages = new Dictionary<string, int>();
ages["Maya"] = 25;
ages.TryGetValue("Leo", out int age);
var tags = new HashSet<string>();
tags.Add("csharp");
tags.Add("csharp"); // one element
```

Dictionary keys are unique; assigning an existing key replaces its value. TryGetValue avoids an exception for a missing key. HashSet stores unique values.

14 LINQ queries

```
int[] nums = { 1, 2, 3, 4 };
var doubled = nums
  .Where(n => n % 2 == 0)
  .Select(n => n * 2)
  .ToList(); // [4, 8]
```

LINQ is in System.Linq. Where filters, Select transforms, OrderBy sorts, Any checks matches. Many queries are deferred; ToList executes and materializes the results.

15 Handle exceptions

```
try {
  int age = int.Parse("oops");
} catch (FormatException error) {
  Console.WriteLine(error.Message);
} finally {
  Console.WriteLine("Finished");
}
```

Catch specific exceptions. finally runs when control leaves the try/catch. Use throw to rethrow the current exception while preserving its stack trace.

16 Read & write files

```
string path = "notes.txt";
File.WriteAllText(path, "Hello\n");
string text = File.ReadAllText(path);
using var reader = new StreamReader(path);
string? line = reader.ReadLine();
```

File and StreamReader are in System.IO. WriteAllText creates or replaces a file. A using declaration disposes the resource at the end of its scope; I/O can throw.

17 Async & await

```
static async Task<string> LoadAsync() {
  return await File.ReadAllTextAsync(
    "notes.txt");
}
string text = await LoadAsync();
```

Task<T> represents an eventual result; Task represents completion. await yields while work is incomplete. It does not automatically create a thread. Prefer await over .Result.

18 Common collection operations

```
var nums = new List<int> { 3, 1, 2 };
nums.Sort();        // [1, 2, 3]
nums.Contains(2);   // true
int size = nums.Count; // 3
nums.Clear();       // empty
```

List<T>, Dictionary<TKey,TValue>, and HashSet<T> live in System.Collections.Generic. Use string.Join(",", values) to join values into text.