UNIT 1
C# Basics

(8 Hours) — Advanced .NET Technology (CACS408)

Note: This PDF may contain errors — please cross-check important points with your syllabus, prescribed textbook and teacher's notes before the exam.

Note: The syllabus line "File 10: File Stream, …" is a typo for "File I/O". Also, "throws" is a Java keyword — C# has only try, catch, finally and throw. Both are handled correctly below.

1.1 Introduction to .NET Architecture

.NET is a free, open-source, cross-platform development platform from Microsoft used to build desktop, web, mobile, cloud and IoT applications. Programs written in a .NET language are not compiled directly into machine code. They are first compiled into an intermediate language (**IL**, also called CIL or MSIL), which the **Common Language Runtime (CLR)** converts to native code at run time using the **JIT** compiler.

bcaink.com

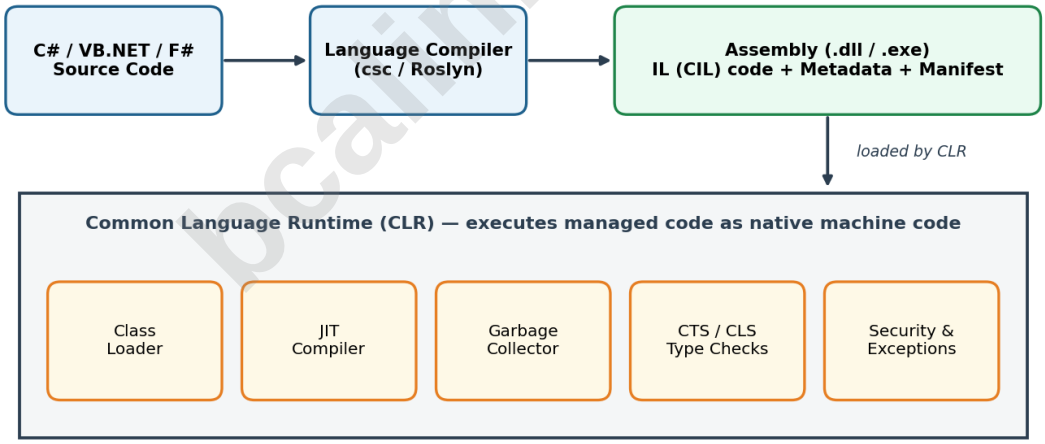


Fig 1.1: .NET compilation and execution model

Core components of the .NET architecture

Component	Meaning / Role
CLR (Common Language Runtime)	Execution engine: loads assemblies, JIT-compiles IL, manages memory (garbage collection), type safety, exceptions, threads and security.
CTS (Common Type System)	Defines all data types and rules (value types, reference types) so that languages can share types.
CLS (Common Language Specification)	A subset of CTS rules every .NET language must follow to be interoperable.
BCL / FCL (Base / Framework Class Library)	Large library of reusable classes: collections, I/O, networking, threading, ADO.NET, ASP.NET, Windows Forms.
IL / MSIL	CPU-independent instruction set produced by the compiler.
JIT Compiler	Converts IL into native machine code the first time a method is called; result is cached.
Assembly	Unit of deployment (.dll or .exe) holding IL code, metadata and a manifest (name, version, references).
Garbage Collector (GC)	Automatically frees memory of objects that are no longer referenced.

boalink.com

Managed vs unmanaged code, .NET Framework vs modern .NET

- **Managed code** runs under the CLR, which provides memory management, security and exception handling. **Unmanaged code** (C/C++ native code) runs directly on the OS.
- **.NET Framework** (versions up to 4.8) is Windows-only; it supports Windows Forms, WPF and classic ASP.NET Web Forms.
- **.NET (formerly .NET Core; .NET 5 and later)** is cross-platform (Windows, Linux, macOS), faster and open-source. ASP.NET Core and Windows Forms (Windows only) run on it.
- **.NET Standard** was a specification of common APIs so a library could be shared by both of the above.

1.2 Class and Object

A **class** is a blueprint that defines data (fields, properties) and behavior (methods, constructors, events). An **object** is an instance of a class created with the `new` keyword. Classes are reference types stored on the heap.

```
using System;

class Student
{
    private string name;            // field
    public int Roll { get; set; }   // auto-implemented property
    public static int Count = 0;    // shared by all objects

    public Student(string name, int roll)   // constructor
    {
        this.name = name;
        Roll = roll;
        Count++;
    }
    public string Name                      // property with logic
    {
        get { return name; }
        set { if (!string.IsNullOrEmpty(value)) name = value; }
    }
    public void Show() => Console.WriteLine($"{Roll} - {name}");
}

class Program
{
    static void Main()
    {
        Student s1 = new Student("Sita", 1);   // creating an object
        s1.Show();                             // 1 - Sita
        Console.WriteLine(Student.Count);      // 1
    }
}
```

Program 1.1: Creating a class and an object

Interface

An **interface** contains only the signatures of members (methods, properties, events) — a contract that implementing classes must fulfil. A class can implement **many interfaces** (this is how C# achieves multiple inheritance of type). Interface members are public by default and an interface cannot be instantiated.

```
interface IShape
{
    double Area();
    string Name { get; }
}

class Circle : IShape
{
    double r;
    public Circle(double r) { this.r = r; }
    public double Area() => Math.PI * r * r;
    public string Name => "Circle";
}

IShape sh = new Circle(2);          // interface reference
Console.WriteLine(sh.Area());       // 12.566...
```

Program 1.2: Creating and implementing an interface

bcalink.com

Access modifiers

Modifier	Accessible from
public	Anywhere, with no restriction.
private	Only inside the same class/struct (default for class members).
protected	Same class and derived classes.
internal	Any code in the same assembly (default for top-level classes).
protected internal	Same assembly OR derived classes in other assemblies.
private protected	Derived classes within the same assembly only.

Arrays

An **array** is a fixed-size collection of elements of the same type, stored contiguously and indexed from 0. Arrays are reference types derived from `System.Array`.

```
int[] a = new int[5];                    // single-dimensional
int[] b = { 10, 20, 30 };                // initializer
int[,] m = new int[2, 3] { {1,2,3}, {4,5,6} };   // rectangular 2-D
int[][] j = new int[2][];                // jagged (array of arrays)
j[0] = new int[] { 1, 2 };
j[1] = new int[] { 3, 4, 5 };

Array.Sort(b);  Array.Reverse(b);
Console.WriteLine(b.Length + " " + m.GetLength(0));   // 3 2
foreach (int x in b) Console.Write(x + " ");
```

Program 1.3: Types of arrays

bcalink.com

Inheritance

Inheritance lets a derived class reuse and extend a base class using the `:` symbol. C# supports single class inheritance (plus many interfaces). Use `virtual` in the base and `override` in the derived class for runtime polymorphism; `abstract` classes cannot be instantiated; `sealed` stops further inheritance; `base` refers to the parent class.

```
class Animal
{
    public string Name;
    public Animal(string n) { Name = n; }
    public virtual void Speak() => Console.WriteLine("Some sound");
}
class Dog : Animal
{
    public Dog(string n) : base(n) { }          // call base constructor
    public override void Speak() => Console.WriteLine(Name + " barks");
}
Animal a = new Dog("Tommy");     // polymorphism
a.Speak();                       // Tommy barks
```

Program 1.4: Inheritance and method overriding

1.3 Exception Handling

An **exception** is a run-time error that disrupts the normal flow of the program. C# handles exceptions with structured blocks. All exceptions derive from `System.Exception` (e.g., `DivideByZeroException`, `FormatException`, `NullReferenceException`, `IndexOutOfRangeException`, `IOException`).

Keyword	Purpose
try	Encloses code that may raise an exception.
catch	Handles a specific exception type; several catch blocks allowed (specific first, general `Exception` last).
finally	Always executes (cleanup such as closing files/connections), whether or not an exception occurred.
throw	Raises an exception manually, or re-throws the current one with `throw;`.
throws	**Not a C# keyword.** It exists in Java; C# has no checked exceptions.

```
try
{
    Console.Write("Enter a number: ");
    int n = int.Parse(Console.ReadLine());
    Console.WriteLine(100 / n);
}
catch (DivideByZeroException) { Console.WriteLine("Cannot divide by zero"); }
catch (FormatException ex)    { Console.WriteLine("Invalid: " + ex.Message); }
catch (Exception ex)          { Console.WriteLine("Error: " + ex.Message); }
finally                       { Console.WriteLine("Done."); }

// user-defined exception
class AgeException : Exception
{
    public AgeException(string msg) : base(msg) { }
}
static void Check(int age)
{
    if (age < 18) throw new AgeException("Must be 18 or older");
}
```

Program 1.5: try-catch-finally, throw and a custom exception

bcalink.com

1.4 Threading

A **thread** is the smallest unit of execution within a process. **Multithreading** lets a program do several things at once (e.g., keep the UI responsive while downloading) and makes better use of multi-core CPUs. Threads live in the `System.Threading` namespace.

Creating a multithreaded program

```
using System;
using System.Threading;

class Program
{
    static void Print(string who)
    {
        for (int i = 1; i <= 3; i++)
        {
            Console.WriteLine(who + " : " + i);
            Thread.Sleep(500);              // WaitSleepJoin state
        }
    }
    static void Main()
    {
        Thread t1 = new Thread(() => Print("Thread-1"));
        Thread t2 = new Thread(() => Print("Thread-2"));
        t1.Start();  t2.Start();            // Unstarted -> Running
        t1.Join();   t2.Join();             // main waits for both
        Console.WriteLine("Main finished");
    }
}
```

Program 1.6: Creating and starting two threads

Thread life cycle

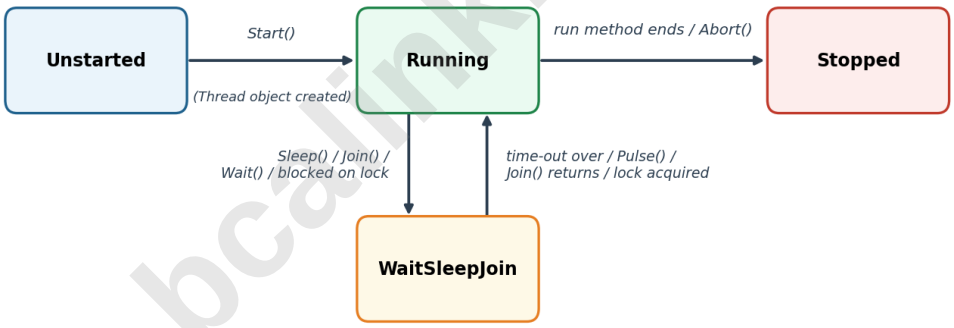


Fig 1.2: Simplified thread life cycle

State (ThreadState)	Description
Unstarted	Thread object created but Start() not yet called.
Running	Started and executing (or ready to run on the CPU).
WaitSleepJoin	Blocked by Sleep(), Join(), Monitor.Wait() or waiting for a lock.
Stopped	Method finished or thread was aborted; cannot be restarted.
Background	Background threads do not keep the application alive; foreground threads do.

Note: Suspend()/Resume() are obsolete and Thread.Abort() is not supported in modern .NET; use flags or CancellationToken to stop a thread safely.

bcalink.com

Thread synchronization (avoiding race conditions)

```
class Counter
{
    static int count = 0;
    static readonly object locker = new object();

    public static void Increment()
    {
        lock (locker)           // only one thread at a time
        {
            count++;
        }
    }
}
```

Program 1.7: lock statement

Other tools are `Monitor`, `Mutex`, `Semaphore`, `Interlocked` and, in modern code, the Task Parallel Library (`Task`, `async/await`).

1.5 File I/O (Files and Streams)

A **stream** is a sequence of bytes flowing between a program and a source such as a file or network. The classes are in `System.IO`. Always wrap them in a `using` block so the file is closed automatically.

Class	Purpose
FileStream	Reads/writes raw bytes in a file. Takes FileMode (Create, Open, Append…), FileAccess and FileShare.
StreamWriter / StreamReader	Write / read **text** (characters, lines) with an encoding.
BinaryWriter / BinaryReader	Write / read primitive types (int, double, string, bool) in **binary** form.
File, FileInfo, Directory	Create, copy, move, delete files and folders; check Exists.

beelink.com

```
// Text file
using (StreamWriter sw = new StreamWriter("note.txt"))
{
    sw.WriteLine("Hello .NET");
    sw.WriteLine("File handling");
}
using (StreamReader sr = new StreamReader("note.txt"))
{
    string line;
    while ((line = sr.ReadLine()) != null) Console.WriteLine(line);
}

// FileStream (bytes)
using (FileStream fs = new FileStream("raw.bin", FileMode.Create, FileAccess.Write))
{
    byte[] data = { 65, 66, 67 };
    fs.Write(data, 0, data.Length);
}

// Binary file
using (BinaryWriter bw = new BinaryWriter(File.Open("d.dat", FileMode.Create)))
{
    bw.Write(101); bw.Write("Ram"); bw.Write(45.5);
}
using (BinaryReader br = new BinaryReader(File.Open("d.dat", FileMode.Open)))
{
    Console.WriteLine(br.ReadInt32() + " " + br.ReadString());
}
```

Program 1.8: StreamWriter/Reader, FileStream, BinaryWriter/Reader

Serialization

Serialization converts an object's state into a stream (binary, XML or JSON) so it can be saved or sent; **deserialization** rebuilds the object. Mark classes with `[Serializable]` (binary) and fields that must not be saved with `[NonSerialized]`.

Type	Class	Remarks
Binary	BinaryFormatter	Classic syllabus topic. **Now obsolete and insecure**; removed from modern .NET (9+). Know it for exams, avoid in real projects.
XML	XmlSerializer	Human-readable; serializes public properties; class needs a parameterless constructor.
JSON	System.Text.Json (JsonSerializer)	Modern, fast, widely used for web APIs.

bcalink.com

```
[Serializable]
class Emp { public int Id; public string Name; [NonSerialized] public string Temp; }

// Binary (old way)
BinaryFormatter bf = new BinaryFormatter();
using (FileStream fs = new FileStream("emp.bin", FileMode.Create))
    bf.Serialize(fs, new Emp { Id = 1, Name = "Hari" });
using (FileStream fs = new FileStream("emp.bin", FileMode.Open))
    Emp e = (Emp)bf.Deserialize(fs);

// JSON (recommended)
using System.Text.Json;
string json = JsonSerializer.Serialize(new Emp { Id = 1, Name = "Hari" });
Emp e2 = JsonSerializer.Deserialize<Emp>(json);
```

Program 1.9: Binary and JSON serialization

Key Takeaways — Unit 1

✓ C# source → IL + metadata (assembly) → JIT compiled by the CLR into native code at run time.

✓ CLR = execution engine; CTS/CLS = type rules; BCL = class library; GC frees memory automatically.

✓ Class = blueprint, object = instance; interface = contract; one base class but many interfaces.

✓ Access modifiers: public, private, protected, internal, protected internal, private protected.

✓ Use virtual/override for polymorphism; abstract and sealed control inheritance.

✓ Exception handling: try, catch, finally, throw. C# has no “throws” keyword.

✓ Thread states: Unstarted → Running ⇄ WaitSleepJoin → Stopped; use lock to synchronize.

✓ File I/O: FileStream (bytes), StreamReader/Writer (text), BinaryReader/Writer (binary); serialization saves objects.

bcalink.com

— End of Unit 1 —