Full Stack Development using .NET

Overview

This document summarizes notes for **Full Stack Development using .NET**. It covers both frontend and backend concepts, then dives into the **C# programming language basics**.
a

1. Frontend

- **HTML**
- **CSS**
- **JavaScript**

2. Backend

- **C#**
- **SQL**
- **ASP.NET**

C# Programming Language Basics

Table of Contents

Introduction

C# is a modern, object-oriented programming language developed by Microsoft. It is type-safe, versatile, and commonly used for desktop, web, and mobile applications.

Some Essential Basics

Before diving deep into C# and .NET, it's important to understand a few building blocks that form the foundation of every project.

Think of a C# application like a city:

- The **Solution** is the city itself — it can hold many buildings (projects).
- Each **Project** is like a building with its own purpose (console app, library, web app, etc.).
- A **Namespace** is like a district in that city, grouping related buildings (classes) together to avoid confusion.
- The **Main() method** acts as the city's power switch — the starting point of the application.
- And finally, the **using directive** is like a shortcut or a map, letting you bring in tools and resources from other parts of the city.

Understanding these fundamentals ensures you can organize, run, and expand your applications smoothly. Now let's break them down one by one:

Solution

- A **Solution** is like a container that can have multiple projects inside.
- Extension: `.sln`

Namespace

- A **Namespace** is like a folder that can contain multiple classes.
- Used to organize code and avoid naming conflicts.

Project Communication

- To make two projects communicate within the same solution:

- Add the project reference of the **library/class project** inside the project that contains the **Program class**.

Main() Method

- Entry point of the application.
- Each project must have only **one Main()** method.

Using

- The `using` directive is used to reference a namespace to use in your project.

Variables

Declaration and Assignment

```
int num;         // Declaration
num = 10;        // Assignment
int num2 = 20;   // Declaration + Assignment
```

Variable Scope

- Variables are accessible only within their scope `{}`.

Example:

```
{
    int num = 5;    // accessible here
    num = 6;
}
// num is NOT accessible here
```

Data Types

Value Types

- Stored in **stack**.
- Examples: `int`, `float`, `char`, `bool`, `struct`, `enum`.

Reference Types

- Reference stored in **stack**, actual value stored in **heap**.
- Examples: `string`, `class`, `interface`, `object`, `array`, `delegate`, `record`.

Strings in C#

Concatenation

```
string s1 = "Amr";
string s2 = "Essam";
string s3 = s1 + " " + s2;  // Concatenation
```

String Interpolation

```
string s1 = "Amr";
string s2 = "Essam";
string s3 = $"{s1} {s2}";   // Interpolation
```

Data Types

Value Data Types

- `sbyte` : 1 byte
- `byte` : 1 byte (unsigned)
- `short` : 2 bytes
- `ushort` : 2 bytes (unsigned)
- `int` : 4 bytes
- `uint` : 4 bytes (unsigned)
- `long` : 8 bytes
- `ulong` : 8 bytes (unsigned)
- `float` : 4 bytes
- `double` : 8 bytes
- `decimal` : 16 bytes
- `char` : 2 bytes (Unicode)
- `bool` : true/false

Var vs Dynamic

`var`

- Type is inferred at **compile time**.
- Once assigned, type **cannot change**.

```
var x = 5;   // int
x = 10;      // valid
// x = "Amr";  // error
```

`dynamic`

- Type resolved at **runtime**.
- Can change data type at any time.

```
dynamic y = 5;    // int
y = "Amr";       // now string
y = true;        // now bool
```

Char vs String

- `char` → single quotes `'A'`, value type.
- `string` → double quotes `"Amr"`, reference type.

Default Values of Data Types

- Reference Types → `null`
- Integers → `0`
- Float/Double/Decimal → `0.0`
- Bool → `false`
- Char → `\0` (null character)
- Struct → all members defaulted
- Enum → first member (default `0`)
- DateTime → `01/01/0001 00:00:00`
- TimeSpan → `00:00:00`

Expressions

- Combination of operands (variables, literals, methods) and operators that evaluate to a single value.

Example

```
var isVip = salary >= 1000;   // returns bool
```

Operators

- **Comparison**: `>`, `<`, `>=`, `<=`, `==`, `!=`
- **Logical**: `&&`, `||`, `!`
- **Bitwise**: `&`, `|`, `^`, `~`, `<<`, `>>`
- **Ternary**: `condition ? value_if_true : value_if_false`
- **Arithmetic**: `+`, `-`, `*`, `/`, `%`, `++`, `--`
- **Assignment**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`

Arrays

Single Dimension Array

```
int[] arr = new int[5];
arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
arr[3] = 4;
arr[4] = 5;  // arr[5] → Error (out of bounds)
```

Initialization with values

```
string[] names = {"Amr", "Ahmed", "Hazem"};
```

Multi-Dimensional Array

```
int[,] multi = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

Console.WriteLine(multi[0,0]); // 1
Console.WriteLine(multi[1,2]); // 6
```

Jagged Array

```
int[][] jagged = {
    new int[] {5, 6, 7},
    new int[] {1, 2, 3},
    new int[] {5}
};
```

Indices and Ranges

```
var arr = new string[5] { "Ahmed", "Amr", "Mohamed", "Eslam", "Mostafa" };
Console.WriteLine(arr[1]);       // "Amr"
var slice = arr[..2];            // ["Ahmed", "Amr"]
var skip = arr[2..];             // ["Mohamed", "Eslam", "Mostafa"]
```

More Expressions

Primary Expression

```
var x = Math.Cos(30) + 1; // returns double
```

Void Expression

```
Console.WriteLine("Hello");
```

Assignment

```
var x = 2;
x = x + 10;
```

Null Coalescing Operator ??

```
string s1 = null;
string s2 = s1 ?? "Amr";   // returns "Amr"
```

Null Conditional Operator ?.

```
string s1 = null;
string s2 = s1?.ToUpper();  // null, no exception
```

Control Statements

If

```
if (condition) {
    Console.WriteLine("True");
}
```

If-Else

```
if (condition) {
    Console.WriteLine("True");
} else {
    Console.WriteLine("False");
}
```

If-ElseIf-Else

```
if (x == 1) {
    Console.WriteLine("One");
} else if (x == 2) {
    Console.WriteLine("Two");
} else {
    Console.WriteLine("Other");
}
```

Switch Case

The switch statement provides a way to execute different blocks of code based on the value of a variable or expression.

General Syntax :

```
switch (expression)
{
    case value1:
        // Code to execute if expression == value1
        break;

    case value2:
        // Code to execute if expression == value2
        break;

    case value3:
        // Code to execute if expression == value3
        break;

    default:
        // Code to execute if none of the above cases match
        break;
}
```

Special case: you can add a multicases on one statement

```
int x = 5;
switch(x) {
    case 1:
        Console.WriteLine("One");
        break;
    case 2:
    case 4:
    case 6:
    case 8:
        Console.WriteLine("Even");
        break;
    case 3:
    case 5:
    case 7:
    case 9:
        Console.WriteLine("Odd");
        break;
    default:
        Console.WriteLine("Zero or Other");
        break;
}
```

Switch Expressions

```
var num = 5;
string res = num switch
{
    1 => "One",
    2 => "Two",
    > 2 => "Greater than Two",
    _ => "Other"
};
```

Looping

While

```
while (condition) {
    // code
}
```

Do-While

```
do {
    // code
} while (condition);
```

For

```
for (int i = 0; i < 10; i++) {
    Console.WriteLine(i);
}
```

Foreach

```
foreach (char c in "Amr") {
    Console.WriteLine(c);
}
```

Infinite Loops

```
while (true) { }
for(;;) { }
```

Jump Statements

- `break` → exit loop
- `continue` → skip to next iteration
- `return` → exit method
- `goto` → jump to label

Casting

Implicit Casting

- Implicit casting happens automatically when converting from a smaller to a larger type

```
int x = 10;
long y = x; // automatic conversion (no data loss)
```

Explicit Casting

- Explicit casting requires manual syntax when converting from a larger to a smaller type

```
long x = 10000;
int y = (int)x; // possible data loss if value doesn't fit in int
```

Boxing

- Boxing: converting a value type (stored in stack) to an object type (stored in heap)

```
int x = 10;
object obj = x; // value -> reference
```

Unboxing

- Unboxing: converting an object back to a value type (requires explicit cast)

```
object obj = 10;
int y = (int)obj;
```

Type Conversion Methods

- Converting between types using built-in methods

```
string str = "10";

int a = int.Parse(str);             // Parse: converts string to int (throws
exception if invalid)
bool ok = int.TryParse(str, out a); // TryParse: safer, returns true/false
instead of throwing
int b = Convert.ToInt32(str);       // Convert: flexible conversion method
```

OOP Concepts

Object-Oriented Programming is a paradigm that allows packaging data and functionality together.

Key principles:

- **Abstraction**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

Classes

```
public class Employee {
    public int Age = 20;
    public string Name;
    public const int Tax = 10;
}
```

Object Creation

```
Employee emp1 = new Employee();
emp1.Age = 25;

int tax = Employee.Tax;
```

Arrays of Objects

```
Employee[] employees = new Employee[2];
employees[0] = emp1;
```

Methods

Basic Methods

```
// A simple method with no parameters and no return value
public void DoSomething() {
    Console.WriteLine("Do something");
}

// A method that returns an integer
public int GetValue() {
    return 10;
}

// A method that takes a parameter
public void PrintAge(int age) {
    Console.WriteLine($"Age: {age}");
}
```

Passing by Reference

```
// Using 'ref' allows the method to modify the original variable
public void Increment(ref int x) {
    x = x + 1;
}

int value = 10;
Increment(ref value);
Console.WriteLine(value); // Output: 11
```

Out Parameters

```
// 'out' requires the method to assign a value before returning
public void Initialize(out int x) {
    x = 10; // Must be set before the method exits
}

int num;
Initialize(out num);
Console.WriteLine(num); // Output: 10
```

Method Overloading

```
// Same method name, but different parameter lists
public void Promote(int amount) {}
public void Promote(int amount, string title) {}
public void Promote(string title, int amount) {}
```

Expression-Bodied Method

```
// Shorter syntax for simple methods (introduced in C# 6.0)
public bool IsEven(int num) => num % 2 == 0;
```

Static Methods

```
// Belongs to the class, not an instance of it
public static void PrintInfo() {
    Console.WriteLine("Static method");
}

// Call without creating an object
Employee.PrintInfo();
```

Constructors

- Used to initialize fields when creating an object.
- Same name as the class, no return type.
- There is a default constructor without parameters.
- You can overload constructors.

Example:

```
public class Date
{
    public int Day;
    public int Month;
    public int Year;

    public Date(int day, int month, int year)
    {
        Day = day;
        Month = month;
        Year = year;
    }
}
```

Readonly Fields

```
public static readonly int X = 5;
```

- Can only be changed inside the constructor.

Properties

- Properties encapsulate private fields with `get` and `set`.
- You can add **validations and constraints**.
- Supports **automatic properties**.
- Can make `set` private to prevent external modification.

Example:

```
public class Student
{
    private string name;
    private int age;
    private char grade;

    public string Name
```

```
    {
        get { return name; }
        set { name = value; }
    }

    // Automatic Property
    public int Age { get; set; }
}
```

Default Properties

- Properties in C# can be **auto-implemented** with a default value.
- This allows assigning an initial value without needing a constructor.

```
class Student
{
    // Auto-implemented property with default value
    public string Name { get; set; } = "Unknown";

    // Another example
    public int Age { get; set; } = 18;
}

class Program
{
    static void Main()
    {
        Student s = new Student();
        Console.WriteLine(s.Name); // Output: Unknown
        Console.WriteLine(s.Age);  // Output: 18
    }
}
```

Indexers

- Allow objects of a class to be accessed like arrays using `[index]`.
- Similar to a property with a parameter.

Syntax:

```
<access_modifier> returnType this[int index]
{
    get { }
    set { }
}
```

Example:

```
public class IP
{
    private int[] segments = new int[4];

    public int this[int index]
    {
        get { return segments[index]; }
        set { segments[index] = value; }
    }
}

// Usage
IP ip = new IP();
ip[0] = 112;
ip[1] = 169;
Console.WriteLine(ip[0]); // 112
```

Notes:

1. You can add constraints like properties.
2. You can make it read-only by removing `set`.
3. Index can be of any type (`int`, `string`, etc.).

Delegates

- A delegate is a type that represents a reference to a method.
- Can be passed as parameters.
- Can store multiple methods.

Syntax:

```
<access_modifier> delegate returnType Name(parameters);
```

Example:

```
public delegate bool IsEven(int number);

public static bool Function(int number)
{
    return number % 2 == 0;
}

public static void Main()
{
```

```
    IsEven isEven = Function;
    Console.WriteLine(isEven(5));
}
```

Adding/Removing Methods

```
isEven += AnotherMethod;
isEven -= AnotherMethod;
```

Passing Delegates as Parameters

```
public delegate bool MyDelegate(Employee e);

public class Employee
{
    public string Name { get; set; }
    public int Age { get; set; }
    public double Salary { get; set; }
}

public class Report
{
    public static void ProcessEmployees(Employee[] emps, string title,
MyDelegate check)
    {
        Console.WriteLine(title);
        foreach (var emp in emps)
        {
            if (check(emp))
                Console.WriteLine(emp.Name);
        }
    }
}
```

Different Delegate Forms

1. **Normal Method**

```
public static bool Check(Employee e) => e.Salary > 5000;
```

2. **Anonymous Method**

```
MyDelegate check = delegate(Employee e) { return e.Salary > 5000; };
```

3. **Lambda Expression**

```
MyDelegate check = (Employee e) => e.Salary > 5000;
```

4. **Simple Lambda**

```
MyDelegate check = e => e.Salary > 5000;
```

Events

- Provide a way to notify other classes when something happens.
- Built on delegates.
- Follows Publisher-Subscriber model.

Syntax:

```
public delegate void Alarm(string message);
public event Alarm OnAlarm;
```

Example:

```
public class Publisher
{
    public delegate void Alarm(string message);
    public event Alarm OnAlarm;

    public void DoSomething()
    {
        Console.WriteLine("Doing something...");
        OnAlarm?.Invoke("Done");
    }
}

// Subscriber
Publisher pub = new Publisher();
pub.OnAlarm += HandleAlarm;

void HandleAlarm(string message)
{
    Console.WriteLine($"Alarm: {message}");
}
```

Unsubscribing:

```
pub.OnAlarm -= HandleAlarm;
```

Operator Overloading

- You can redefine operators (`+`, `-`, `==`, etc.) for custom types.

Rules:

1. Must be `static` and `public`.
2. Can only overload existing operators.
3. If you overload `>`, you must also overload `<`.

Example:

```
public class Complex
{
    public double Real { get; set; }
    public double Imag { get; set; }

    public Complex(double real, double imag)
    {
        Real = real;
        Imag = imag;
    }

    public static Complex operator +(Complex a, Complex b)
    {
        return new Complex(a.Real + b.Real, a.Imag + b.Imag);
    }
}
```

Finalizers

- Destructor of a class (opposite of constructor).
- Called before object is reclaimed by the Garbage Collector.

Syntax:

```
~ClassName()
{
    // cleanup code
}
```

Rules:

1. Cannot have parameters.
2. Called non-deterministically by GC.

Garbage Collector

- Automatically manages memory.
- Reclaims objects no longer referenced.
- Manual collection possible via `GC.Collect()`, but not recommended.

Structs

Structs are **value types** (stored on the stack in most cases). They are best used for small and lightweight objects.

```
struct Point
{
    public int X;
    public int Y;
}
```

Differences Between Class and Struct

Feature	Struct	Class
Type	Value	Reference
Memory Allocation	Stack	Heap
Inheritance	Cannot inherit (except from `ValueType`)	Can inherit from another class
Default Constructor	Only implicit, cannot define a parameterless one	Allowed
Finalizer	Not allowed	Allowed
Null Value	Cannot be null	Can be null

👉 Default behavior:

```
Point point = new Point(); // X = 0, Y = 0 (default values)
```

Members Allowed in Structs

1. Fields (cannot be initialized inside struct definition)
2. Properties
3. Methods
4. Events
5. Indexers
6. Operators
7. Constructors (only with parameters)
8. Constants (must be initialized)

Access Modifiers in Structs

- Members can be `public`, `private`, or `internal`. By default, they are `private`.

Notes on Structs

1. Keep struct size small (preferred < 16 bytes).
2. Use `readonly` struct for immutability.
3. Use properties instead of fields.
4. To pass by reference, use `out` or `ref`.

Enums

Enums represent a set of **named constants**.

```
enum Days
{
    Saturday = 10,
    Sunday = 11,
    Friday = 16
}
```

- Values start from `0` by default and increment by `1`.
- You can change the starting value by assigning explicitly.

Changing the Underlying Type

By default, `enum` is of type `int`. You can specify another integral type:

```
enum Days : byte
{
    Monday, Tuesday, Wednesday
}
```

Allowed underlying types: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`.

Accessing Enum Values

```
Console.WriteLine(Days.Monday);   // Monday
Console.WriteLine((int)Days.Monday); // 0

Days d = (Days)3; // Casting integer to enum

string s = "Friday";
Days d2 = (Days)Enum.Parse(typeof(Days), s);
```

Enum Methods

- `ToString()` → returns the name as string
- `Enum.GetNames(typeof(Days))` → returns all names as string[]
- `Enum.GetValues(typeof(Days))` → returns all values as enum[]
- `Enum.IsDefined(typeof(Days), "Sunday")` → checks if a value exists

Flags Enum

Used for bitwise combinations.

```
[Flags]
enum Permission
{
    Read = 1,       // 0001
    Write = 2,      // 0010
    Execute = 4     // 0100
}

Permission p = Permission.Read | Permission.Write; // Combination using bitwise
OR
```

Inheritance

Inheritance allows a class to reuse members of another class.

```
class Animal
{
    public void Walk() => Console.WriteLine("Animal Walking...");
}

class Dog : Animal
{
    public void Bark() => Console.WriteLine("Bark!");
}
```

Rules

- C# supports **single inheritance** (one base class).
- A class can inherit **one class + multiple interfaces**.

Access Modifiers

- `public`: accessible everywhere
- `protected`: accessible in base and derived classes
- `private`: accessible only in base class (not inherited)

Calling Base Constructor

```
class Base
{
    public int X, Y;
    public Base(int x, int y)
    {
        X = x;
        Y = y;
    }
}

class Child : Base
{
    public Child(int x, int y) : base(x, y) { }
}
```

Abstract Classes

- Declared with the `abstract` keyword.
- Cannot be instantiated directly.
- Designed to be base classes.

Can contain:

- **Abstract members**: no implementation, must be overridden
- **Concrete members**: normal methods, fields, properties

```
abstract class Shape
{
    public abstract double Area();
}

class Circle : Shape
{
```

```
    public double Radius;
    public Circle(double r) => Radius = r;
    public override double Area() => Math.PI * Radius * Radius;
}
```

Sealed Classes

A **sealed class** cannot be inherited.

```
sealed class MyClass { }
```

Polymorphism

Polymorphism = many forms → allows the same method to behave differently depending on the object.

Types of Polymorphism

1. **Compile-time (Static):**
    - Method Overloading
    - Operator Overloading

```
class Calc
{
    public int Add(int x, int y) => x + y;
    public double Add(double x, double y) => x + y;
}
```

2. **Run-time (Dynamic):**
    - Method Overriding using `virtual` and `override`

```
class Animal
{
    public virtual void Speak() => Console.WriteLine("Some sound");
}

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

Virtual vs New

```
class Base
{
    public virtual void Show() => Console.WriteLine("Base Show");
}

class Derived : Base
{
    public override void Show() => Console.WriteLine("Derived Show");
}

class Hiding : Base
{
    public new void Show() => Console.WriteLine("Hidden Show");
}

Base b1 = new Derived();
Base b2 = new Hiding();

b1.Show(); // Derived Show
b2.Show(); // Base Show

Hiding h = new Hiding();
h.Show();  // Hidden Show
```

Interfaces

- Defines a **contract** for classes/structs.
- Can contain **method/property/indexer/event signatures**.
- No fields allowed.
- Members are **public by default**.
- A class/struct can implement multiple interfaces.
- From C# 8+, interfaces can have **default implementations**.

```
interface IAnimal
{
    void Speak();
}

class Dog : IAnimal
{
    public void Speak() => Console.WriteLine("Bark");
}

IAnimal dog = new Dog();
dog.Speak();
```

⚠ Cannot instantiate an interface:

```
IAnimal animal = new IAnimal(); // ❌ Not allowed
```