

UNIT 2

The C# Language Basics

(12 Hours) — DotNet Technology (CAC302)

2.1 Writing Console and GUI Applications

C# programs can be built as Console Applications, which run inside a text-based terminal window and communicate through standard input/output, or as GUI (Graphical User Interface) Applications, which use visual controls such as windows, buttons and text boxes (typically built with Windows Forms or WPF). Both types are compiled and executed by the .NET Common Language Runtime (CLR), but they differ in how the user interacts with them.

Aspect	Console Application	GUI Application
Interaction	Text-based, via keyboard/terminal	Visual, via mouse/keyboard on forms
Entry Point	Main() method	Main() → Application.Run(new Form1())
Output	Console.WriteLine()	Controls: Label, TextBox, Button, etc.
Typical Use	Utilities, learning, batch tasks	Desktop software with rich UI

Example 2.1 — A Simple Console Application

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

Explanation

The Main() method is the entry point of every C# console application. The compiler looks for this method to start execution. Console.WriteLine() prints text followed by a newline to the terminal window.

A minimal GUI application in Windows Forms replaces the console output with a window containing controls:

Example 2.2 — A Simple Windows Forms Skeleton

```
static class Program {  
    static void Main() {  
        Application.EnableVisualStyles();  
        Application.Run(new MainForm());  
    }  
}
```

Note: Application.Run() starts a standard application message loop that keeps the window responsive to user events such as clicks and key presses.

2.2 Identifiers and Keywords

An identifier is the name a programmer gives to a program element such as a class, variable, method, or namespace. A keyword is a word reserved by the C# language itself and carries special meaning to the compiler; it cannot be used as an ordinary identifier.

Rules for Identifiers

- Must begin with a letter or an underscore (_), never with a digit.
- May contain letters, digits, and underscores after the first character.
- Are case-sensitive — Total and total are different identifiers.
- Cannot be the same as a reserved keyword unless prefixed with @ (a verbatim identifier), e.g. @class.
- Should be meaningful and follow naming conventions (PascalCase for classes/methods, camelCase for local variables).

Categories of Keywords

Category	Examples
Reserved keywords	class, if, else, for, while, return, namespace, public
Contextual keywords	var, yield, partial, async, await, get, set

Note: Contextual keywords behave as keywords only in specific code contexts; elsewhere they may be used as ordinary identifiers.

2.3 Writing Comments

Comments are non-executable notes inserted into source code to explain logic, improve readability, and assist future maintenance. The compiler ignores them entirely.

Type	Syntax	Purpose
Single-line	// comment text	Brief note on one line
Multi-line	/* comment text */	Block explanation spanning several lines
XML documentation	/// <summary> ... </summary>	Generates IntelliSense / API docs

Example 2.3 — Comment Styles

// This calculates the area of a circle

```
double radius = 5.0;
```

/* The formula used below is

Area = $\pi * r * r$ */

```
double area = Math.PI * radius * radius;
```

/// <summary>Returns the square of a number.</summary>

```
int Square(int n) { return n * n; }
```

2.4 Data Types

C# is a strongly typed language — every variable and expression has a type known at compile time. Data types are grouped into value types, which store data directly, and reference types, which store a reference (address) to the data on the heap.

Value Types

Type	Size	Range / Description
int	4 bytes	-2,147,483,648 to 2,147,483,647
long	8 bytes	Large whole numbers
float	4 bytes	~6-7 significant digits, suffix F
double	8 bytes	~15-16 significant digits (default for decimals)
decimal	16 bytes	High precision, used for currency, suffix M
char	2 bytes	A single Unicode character, e.g. 'A'
bool	1 byte	true or false

Reference Types

- string — an immutable sequence of characters.
- object — the base type from which all other types derive.
- arrays, classes, interfaces, delegates — user-defined or built-in composite types.

Example 2.4 — Declaring Variables of Different Types

```
int age = 21;  
double price = 499.95;  
char grade = 'A';  
bool isPassed = true;  
string name = "Sujan";
```

Note: Value types are stored on the stack (fast, automatic cleanup); reference types are stored on the heap and managed by the garbage collector.

2.5 Expressions and Operators

An expression is a combination of operands (variables, literals, method calls) and operators that evaluates to a single value. C# provides a rich set of operators grouped by function.

Category	Operators	Example
Arithmetic	+ - * / %	a + b, a % b
Relational	== != < > <= >=	a >= b
Logical	&& !	(a > 0) && (b > 0)
Assignment	= += -= *= /=	total += price
Increment/Decrement	++ --	count++, --index
Conditional (ternary)	condition ? expr1 : expr2	max = (a > b) ? a : b
Bitwise	& ^ ~ << >>	flags & mask

Example 2.5 — Using the Ternary Operator

```
int a = 15, b = 20;  
int max = (a > b) ? a : b;  
Console.WriteLine("Maximum is: " + max);
```

Output

Maximum is: 20

*Note: Operator precedence determines the order of evaluation — e.g. * and / bind tighter than + and -. Use parentheses to make intent explicit.*

2.6 Strings and Characters

The `char` type represents a single 16-bit Unicode character enclosed in single quotes, e.g. `'A'`. The `string` type represents an immutable sequence of characters enclosed in double quotes. Because strings are immutable, every modification (such as concatenation) actually creates a new string object in memory.

Common String Operations

Operation	Method / Syntax	Result Example
Concatenation	<code>s1 + s2</code> or <code>string.Concat()</code>	<code>"Hello" + " World"</code>
Length	<code>s.Length</code>	<code>"Hello".Length → 5</code>
Substring	<code>s.Substring(0, 3)</code>	<code>"Hello".Substring(0,3) → "Hel"</code>
Upper/Lower case	<code>s.ToUpper()</code> / <code>s.ToLower()</code>	<code>"Hi".ToUpper() → "HI"</code>
Compare	<code>string.Compare(s1, s2)</code>	returns -1, 0, or 1
Format / Interpolation	<code>\$"Hi {name}"</code>	<code>"Hi Ram"</code>

Example 2.6 — String Interpolation

```
string firstName = "Anisha";  
int marks = 87;  
string message = $"{firstName} scored {marks} marks.";   
Console.WriteLine(message);
```

Output

Anisha scored 87 marks.

Note: Because string objects are immutable, repeated concatenation in a loop is inefficient — use the `StringBuilder` class for building large strings incrementally.

2.7 Arrays

An array is a fixed-size, ordered collection of elements of the same type, stored contiguously and accessed using a zero-based index. C# supports single-dimensional, multidimensional (rectangular), and jagged arrays.

Array Type	Declaration Example	Description
Single-dimensional	<code>int[] nums = new int[5];</code>	A simple linear list of elements
Multidimensional	<code>int[,] grid = new int[3,3];</code>	A rectangular table (rows × columns)
Jagged	<code>int[][] jag = new int[3][];</code>	An array of arrays — rows can differ in length

Example 2.7 — Declaring and Using a One-Dimensional Array

```
int[] marks = { 78, 85, 92, 60, 74 };
int sum = 0;
for (int i = 0; i < marks.Length; i++)
    sum += marks[i];
Console.WriteLine("Total = " + sum);
```

Output

Total = 389

Note: Array indices in C# always start at 0; accessing marks[5] on a 5-element array throws an `IndexOutOfRangeException`.

2.8 Variables and Parameters

A variable is a named storage location whose value can change during program execution. Declaring a variable requires specifying its type (or using `var` for compiler-inferred typing) followed by an identifier.

Example 2.8 — Variable Declaration and Type Inference

```
int quantity = 10; // explicitly typed
var price = 25.50; // compiler infers 'double'
const double TaxRate = 0.13; // constant, cannot change
```

Parameters

Parameters are variables listed in a method's signature that receive values (arguments) when the method is called. C# supports several parameter-passing mechanisms:

Modifier	Behaviour
(none) — by value	A copy of the argument is passed; changes inside the method do not affect the original.

Modifier	Behaviour
ref	Passes by reference; the variable must be initialized before the call, and changes affect the caller's variable.
out	Passes by reference specifically to return a value; need not be initialized before the call.
params	Allows a variable number of arguments to be passed as an array.

Example 2.9 — ref and out Parameters

```
void Square(ref int n) { n = n * n; }

bool TryDivide(int a, int b, out int result) {
    if (b == 0) { result = 0; return false; }
    result = a / b;
    return true;
}
```

Usage

```
int x = 5; Square(ref x); // x becomes 25
int q; bool ok = TryDivide(10, 2, out q); // q = 5, ok = true
```

2.9 Statements

A statement is the smallest executable unit of a C# program — it performs an action and ends with a semicolon (except block statements). C# statements fall into five broad categories.

Category	Purpose	Examples
Declaration	Introduces a variable or constant	int x; const double PI = 3.14;
Expression	Evaluates an expression, often with a side effect	x = 5; Console.WriteLine(x);
Selection	Chooses among alternative execution paths	if / else, switch
Iteration	Repeats a block of statements	for, while, do-while, foreach
Jump	Transfers control elsewhere in the program	break, continue, return, goto

2.9.1 Selection Statements

Example 2.10 — if / else and switch

```
int marks = 72;
if (marks >= 90) Console.WriteLine("Grade A");
else if (marks >= 70) Console.WriteLine("Grade B");
else Console.WriteLine("Grade C");

switch (marks / 10) {
    case 9: case 10: Console.WriteLine("A"); break;
    case 7: case 8: Console.WriteLine("B"); break;
    default: Console.WriteLine("C"); break;
}
```

Output

Grade B

B

2.9.2 Iteration Statements

Example 2.11 — for, while, and foreach

```
for (int i = 1; i <= 3; i++) Console.Write(i + " "); // 1 2 3

int n = 3;
while (n > 0) { Console.Write(n + " "); n--; } // 3 2 1

int[] arr = { 4, 5, 6 };
foreach (int v in arr) Console.Write(v + " "); // 4 5 6
```

2.9.3 Jump Statements

Statement	Effect
break	Exits the nearest enclosing loop or switch statement.
continue	Skips to the next iteration of the enclosing loop.
return	Exits the current method, optionally returning a value.
goto	Transfers control to a labeled statement (rarely used).

Note: Prefer for/foreach over goto for readability; goto is retained mainly for compatibility and specific low-level control-flow cases such as breaking out of nested loops via a label.

2.10 Namespaces

A namespace is a logical container used to organize related classes, interfaces, and other types, and to prevent naming conflicts between libraries that might otherwise define types with identical names. The .NET Framework Class Library itself is organized entirely using namespaces, e.g. System, System.Collections.

Example 2.12 — Defining and Using a Namespace

```
namespace SchoolApp {  
    class Student {  
        public string Name;  
    }  
}  
  
// In another file:  
using SchoolApp;  
class Program {  
    static void Main() {  
        Student s = new Student();  
        s.Name = "Kritika";  
    }  
}
```

Note: The using directive simply saves typing the fully qualified name (e.g. SchoolApp.Student); it does not import code physically, only brings the namespace into scope.

Key Takeaways

- ✓ C# supports both console applications (text I/O) and GUI applications (visual controls).
- ✓ Identifiers name program elements; keywords are reserved words with special compiler meaning.
- ✓ Comments (//, /* */, ///) document code without affecting execution.
- ✓ Data types split into value types (stack) and reference types (heap).
- ✓ Operators combine with operands to form expressions following defined precedence rules.
- ✓ Strings are immutable; chars represent single Unicode characters.
- ✓ Arrays store fixed-size, same-type collections — single, multidimensional, or jagged.
- ✓ Variables hold data; parameters (value, ref, out, params) control how data moves into methods.
- ✓ Statements are categorized as declaration, expression, selection, iteration, and jump.
- ✓ Namespaces organize types logically and avoid naming collisions across libraries.

— *End of Unit 2* —

bcalink.com