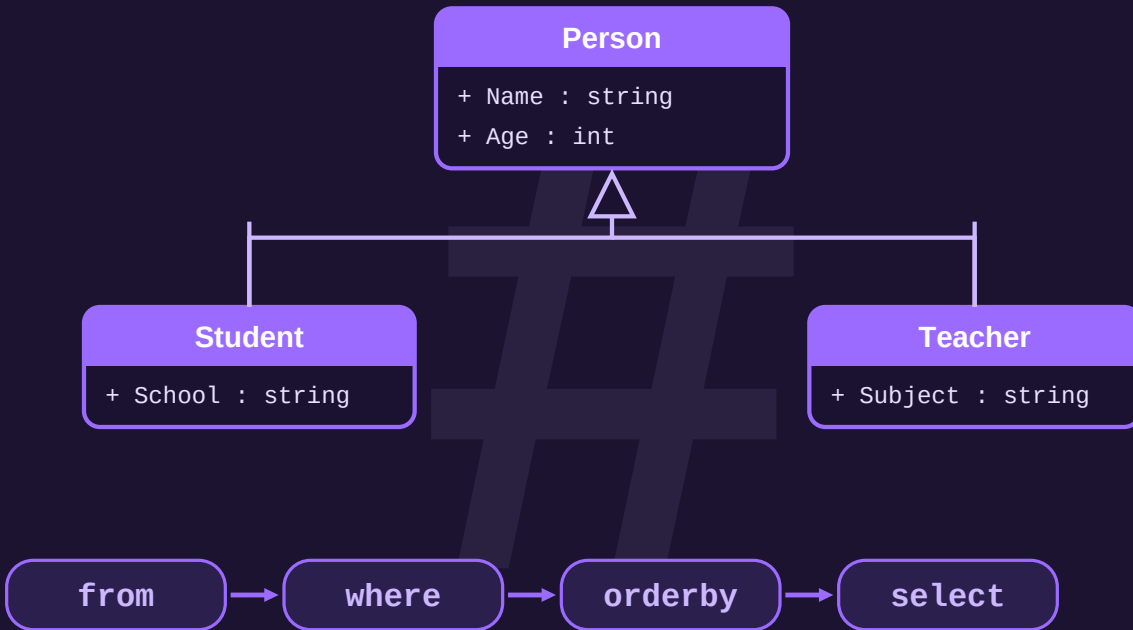




BioGlytch Studios LLC

Learn | Build | Grow



Classes, inheritance and LINQ queries

C# PROGRAMMING

COMPUTER CODING: LEARN TO PROGRAM

WRITTEN BY

Clarence Scott & BioGlytch Studios

BioGlytch Studios LLC | Est. 2026

In Partnership with Gloria Coles Flint Public Library

bioglytchstudios.tech | 2026 Edition

C# Programming

Computer Coding: Learn to Program

BioGlytch Studios LLC

COPYRIGHT 2026 © BioGlytch Studios LLC © Genesee County, MI EST 2026

</HELLO CODER>

Welcome to the engineering world of BioGlytch Studios LLC! We are thrilled to present this book to you. Published by BioGlytch Studios LLC and offered as a front-facing resource of Gloria Coles Flint Public Library, it is dedicated to making computer science and technology education more accessible to our community. Designed for aspiring technology enthusiasts and their supportive parents, this book offers an opportunity to explore the exciting world of computing together. If you are a parent who has chosen this book to introduce your child to technology, we thank you for your trust and support.

Through this book, and our partnership with Gloria Coles Flint Public Library, BioGlytch Studios LLC aims to ignite curiosity and inspire a passion for coding, programming, and computing technologies in individuals of all ages. We recognize that access to quality educational resources can be limited by financial barriers, time constraints, and the rising costs of traditional university courses. Our goal goes beyond teaching the fundamentals—we want to empower you to think critically, solve problems creatively, and begin your journey toward becoming a skilled and innovative engineer.

Throughout this book, you will develop essential skills such as critical thinking, logical outlooks, problem solving, discipline and career-ready structure. Together, we are committed to building a stronger, more technologically empowered community, one coder at a time.

Each successful assessment will grant a Certificate from BioGlytch Studios LLC, along with a PDF of your course completion to show your next employer.

Thank you for joining us on this journey. We look forward to supporting your growth and celebrating your progress as you put your best code forward!

Sincerely,

Clarence Scott

CEO & Founder, BioGlytch Studios LLC

In Partnership with Gloria Coles Flint Public Library

Publishing Information

Book Title

C# Programming

Author

Clarence Scott

Publisher

BioGlytch Studios LLC

2026 Edition

Copyright Notice

© 2026 BioGlytch Studios LLC. All rights reserved.

BioGlytch Studios LLC is the sole owner of all content in this book. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Library Partnership

Gloria Coles Flint Public Library offers this book as a front-facing resource for its organization. The partnership does not transfer ownership of any content, which remains solely with BioGlytch Studios LLC.

Edition

First Edition

Disclaimer

The information contained in this book is provided for educational purposes only. While every effort has been made to ensure accuracy, the author and publisher are not responsible for errors, omissions, or damages resulting from the use of this material.

Printing Information

Printed in Genesee County, Michigan

Printed by BioGlytch Studios LLC

Contact Information

For inquiries, feedback, or bulk orders, please contact:

Email: software@bioglytchstudios.tech

Website: <https://bioglytchstudios.tech>

Phone: (989) 900-6747

Table of Contents

Overview

- Overview
- Conclusion

Chapter 1: Introduction to C# Programming

- Introduction to C#
- The Structure of a C# Program
- Writing, Compiling, and Running a C# Program

Chapter 2: Data Types, Variables and Operators

- Primitive Data Types
- Variable Declaration and Initialization
- Operators in C#
- Type Conversion

Chapter 3: Control Flow Statements

- Conditional Statements
- Looping Statements
- Jump Statements

Chapter 4: Methods and Functions

- Method Declaration and Invocation
- Method Overloading
- Parameter Passing
- Return Types

Chapter 5: Object-Oriented Programming (OOP) Basics

- Classes and Objects
- Fields, Properties, and Methods
- Constructors and Destructors
- Access Modifiers
- Inheritance and Polymorphism

Chapter 6: Arrays and Collections

- Single-Dimensional and Multi-Dimensional Arrays
- Collections in C#
- Basic Operations on Collections

Chapter 7: Exception Handling

- Try, Catch, Finally Blocks
- Throwing Exceptions
- Custom Exceptions

Chapter 8: File I/O and Streams

- Reading from and Writing to Files
- Working with Streams
- Handling File Exceptions

Chapter 9: Basic Understanding of LINQ

- Introduction to LINQ
- Basic LINQ Queries
- Using LINQ with Collections

Chapter 10: Basic Concepts of Debugging and Testing

- Debugging Techniques
- Writing Unit Tests

Chapter 11: Introduction to Visual Studio

- Overview of the Visual Studio IDE
- Creating, Building, and Running a C# Project
- Using the Debugger

Modern C# for 2026

- Versions at a Glance
- A Modern Project File
- What's New in C# 14
- Async and Await
- A Tiny Web API

2026 C# Best Practices

Appendix

- Appendix A: C# Programming Environment Setup
- Appendix B: Key C# Syntax and Commands
- Appendix C: LINQ Examples

Glossary

Overview

Overview

Chapter Summaries

Chapter 1: Introduction to C# Programming This chapter introduces the basics of C# programming, including its history, key features, and the structure of a C# program. You'll learn how to write, compile, and run your first C# program, gaining an understanding of the development workflow.

Chapter 2: Data Types, Variables, and Operators Explore the fundamental building blocks of C# programming: data types, variables, and operators. This chapter covers primitive data types, variable declaration and initialization, various operators, and type conversion techniques.

Chapter 3: Control Flow Statements Learn how to control the flow of your programs using conditional statements, looping constructs, and jump statements. This chapter explains if-else conditions, switch cases, for loops, while loops, and the use of break and continue statements.

Chapter 4: Methods and Functions Delve into methods and functions, essential for organizing and modularizing your code. Topics include method declaration, invocation, overloading, parameter passing, and return types, providing you with the skills to create reusable and maintainable code.

Chapter 5: Object-Oriented Programming (OOP) Basics Understand the principles of object-oriented programming, a core concept in C#. This chapter covers classes, objects, fields, properties, methods, constructors, destructors, access modifiers, inheritance, and polymorphism, enabling you to design robust and scalable applications.

Chapter 6: Arrays and Collections Master the use of arrays and collections to handle groups of data efficiently. You'll learn about single-dimensional and multi-dimensional arrays, and explore collections such as lists, dictionaries, and other key-value pair structures.

Chapter 7: Exception Handling Gain knowledge on how to handle runtime errors gracefully using exception handling mechanisms. This chapter covers try, catch, and finally blocks, throwing exceptions, and creating custom exceptions to ensure your programs can manage unexpected situations.

Chapter 8: File I/O and Streams Learn to read from and write to files using streams. This chapter provides an overview of file handling in C#, including reading from and writing to text files, working with streams, and handling file-related exceptions for robust file I/O operations.

Chapter 9: Basic Understanding of LINQ Discover the power of Language Integrated Query (LINQ) for querying collections in C#. This chapter introduces LINQ syntax, basic LINQ queries, and demonstrates how LINQ simplifies data manipulation compared to traditional

looping constructs.

Chapter 10: Basic Concepts of Debugging and Testing Understand essential debugging techniques and the importance of unit testing. This chapter covers the use of breakpoints, step-through debugging, and writing unit tests to ensure your code is reliable and free of bugs.

Chapter 11: Introduction to Visual Studio Get familiar with Visual Studio, the powerful IDE for C# development. Learn how to create, build, and run C# projects, and utilize the debugger to troubleshoot and refine your applications.

End-of-Chapter Exercises

Each chapter includes multiple-choice questions to test your theoretical understanding, coding tasks to practice hands-on programming, and problem-solving scenarios to apply your knowledge in real-world situations.

Appendices and Glossary

The appendices provide additional resources, such as setting up your C# programming environment, key syntax and commands, and LINQ examples. The glossary defines key terms and concepts to reinforce your understanding of C# programming.

Conclusion

"C#:Basics to Advanced" equips you with the knowledge and skills to develop robust, efficient, and scalable applications in C#. Through practical exercises and real-world scenarios, this book ensures you are well-prepared to tackle challenges in the field of software development. Whether you're aiming to enhance your career prospects or build innovative projects, this comprehensive guide is your gateway to mastering C#.

Chapter 1: Introduction to C# Programming

Introduction to C#

C# (pronounced as "C sharp") is a modern, versatile programming language developed by Microsoft. It is widely used for building various types of applications, from desktop software to web applications and games. C# is known for its simplicity, type-safety, and strong support for object-oriented programming concepts.

The Structure of a C# Program

A C# program is structured into classes, methods, statements, expressions, and variables. At its core, every C# application starts with a Main method, which serves as the entry point for the program execution. Here's a basic structure of a simple C# program that prints "Hello, World!":

```
Console.WriteLine("Hello, World!");
```

That is a complete program. Since C# 9, **top-level statements** let you write your code without a class or a Main method, and projects created with the .NET 6 and newer templates include the common using directives automatically (called implicit usings). The compiler still creates a Main method for you behind the scenes. The longer, traditional form below does exactly the same thing and is still common in larger programs, so you should be able to read both:

```
namespace HelloApp;

internal class Program
{
    private static void Main()
    {
        Console.WriteLine("Hello, World!");
    }
}
```

2026 Best Practice: Each example in this book is written as a complete `Program.cs`. When you try a new example, create a fresh project or replace the contents of `Program.cs`. In a file that uses top-level statements, your statements must come first, and any classes or records you declare go below them.

Writing, Compiling, and Running a C# Program

To write a C# program, you typically use an Integrated Development Environment (IDE) such as Visual Studio, Visual Studio Code, or JetBrains Rider. These IDEs provide a user-friendly interface for writing code, managing projects, and debugging applications.

After writing your C# code, the compiler turns it into an executable format. Today you do not need to call the compiler yourself. The **.NET SDK** (we use **.NET 10**, a Long-Term Support release that comes with **C# 14**) builds and runs your project with the `dotnet` command-line tool:

```
dotnet new console -n HelloApp    # create a new console project
cd HelloApp
dotnet run                        # compile and run
dotnet build                      # compile only
```

Visual Studio 2026, Visual Studio Code and JetBrains Rider run the same commands for you when you press the Run button. In .NET 10 you can even run a single file, with no project, by saving your code as `hello.cs` and typing `dotnet run hello.cs`.

Once compiled, you can run the C# program either from within the IDE or from the command line. Running the program executes the instructions you've written, producing the desired output or performing the intended tasks.

End-of-Chapter Exercises:

Multiple-Choice Questions:

1. What is the correct syntax to write a "Hello, World!" program in C#?
 - A) `print("Hello, World!");`
 - B) `System.out.println("Hello, World!");`
 - C) `Console.WriteLine("Hello, World!");`
 - D) `writeLine("Hello, World!");`
2. Which IDE is commonly used for C# development?
 - A) Eclipse
 - B) IntelliJ IDEA
 - C) Visual Studio
 - D) Sublime Text

Coding Tasks:

1. Write a C# program that prints "Welcome to C# Programming!" to the console.

```
Console.WriteLine("Welcome to C# Programming!");
```

Problem-Solving Scenarios:

1. Explain how you would set up your development environment for C# programming.

- To set up a development environment for C# programming, follow these steps:
- **Choose an IDE:** Select an IDE such as Visual Studio 2026, Visual Studio Code, or JetBrains Rider.
- **Install .NET SDK:** Download and install the .NET Software Development Kit (SDK) from the official .NET website.
- **Create a Project:** Use your chosen IDE to create a new C# project. Choose the type of application you want to build (e.g., Console Application, Web Application).
- **Write and Run Code:** Write your C# code in the IDE. Use the IDE's build and run commands to compile and execute your program.
- **Debugging:** Learn to use the debugging tools provided by the IDE to identify and fix errors in your code.

Chapter 2: Data Types, Variables and Operators

Primitive Data Types

In C#, primitive data types represent the most basic types of data that the language supports. These include:

- **Integer Types:** int , long , short , byte , sbyte , uint , ulong , ushort
- **Floating-Point Types:** float , double , decimal
- **Character Type:** char
- **Boolean Type:** bool

Each data type has specific ranges and uses, depending on the size and precision required for storing data.

Three more types you will use constantly are string (text), object (anything) and decimal (exact base-10 numbers, which you should use for money). Let the compiler work out the type with var when the type is obvious from the right-hand side, and use const for values that never change:

```
const decimal TaxRate = 0.07m;    // the m suffix means decimal
var price = 19.99m;               // decimal
var name = "Ava";                 // string
var count = 3;                    // int
decimal total = price * count * (1 + TaxRate);
Console.WriteLine($"Total: {total:C}"); // formatted as currency
```

2026 Best Practice: New projects have **nullable reference types** turned on. A string is not allowed to be null; write string? when a value might be missing. The compiler then warns you before a missing value can crash your program.

```
string? middleName = null;        // allowed to be null
Console.WriteLine(middleName?.Length ?? 0); // safe: prints 0
```

Variable Declaration and Initialization

Variables in C# are containers that hold data of a specific type. To declare a variable, you specify the type followed by the variable name. Variables must be initialized before they can be used.

```
int age; // Declaration
age = 30; // Initialization
```

You can also declare and initialize a variable in a single statement:

```
double pi = 3.14;
```

Operators in C#

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

- **Arithmetic Operators:** + , - , * , / , %
- **Relational Operators:** == , != , < , > , <= , >=
- **Logical Operators:** && , || , !
- **Assignment Operators:** = , += , -= , *= , /= , %=
- **Increment and Decrement Operators:** ++ , --
- **Bitwise Operators:** & , | , ^ , ~ , << , >>

Type Conversion

Type conversion (also known as casting) in C# is the process of converting a value from one data type to another. Implicit conversion is done automatically by the compiler when there is no risk of data loss. Explicit conversion requires explicit casting and may involve data loss.

```
int num1 = 10;
double num2 = 5.5;
int result = num1 + (int)num2; // Explicit casting from double to int
```

Never trust text typed by a user. `Convert.ToInt32` and `int.Parse` crash on bad input. Use `TryParse`, which tells you whether the conversion worked:

```
Console.Write("Enter your age: ");
string? input = Console.ReadLine();

if (int.TryParse(input, out int age))
{
    Console.WriteLine($"Next year you will be {age + 1}.");
}
else
{
    Console.WriteLine("That is not a whole number.");
}
```

End-of-Chapter Exercises:

Multiple-Choice Questions:

1. What is the default value of an `int` variable in C#?

- A) 0
- B) 1
- C) null

- D) undefined 2. Which operator is used for division in C#?
- A) +
- B) -
- C) *
- D) /

Coding Tasks:

1. Write a C# program to swap the values of two variables.

```
int a = 10;
int b = 20;
Console.WriteLine($"Before swapping: a = {a}, b = {b}");

// Swap logic with a temporary variable
int temp = a;
a = b;
b = temp;
Console.WriteLine($"After swapping: a = {a}, b = {b}");

// Modern shortcut: swap with a tuple, no temp variable needed
(a, b) = (b, a);
Console.WriteLine($"Swapped back: a = {a}, b = {b}");
```

Problem-Solving Scenarios:

1. Describe a scenario where type conversion might be necessary and how you would handle it in C#.

- **Scenario:** You have an application that calculates the total price of items in a shopping cart. The price of each item is stored as a double, but you need to display the total price as an int (assuming you want to round down to the nearest integer).
- **Handling Type Conversion:** In this scenario, you would use explicit casting ((int)) to convert the double total price to an int :

```
double totalPrice = 49.99; // Example total price
int totalAsInt = (int)totalPrice;
```

2026 Best Practice: Money should be stored in decimal, not double, because double cannot represent values like 0.1 exactly. Use `Math.Round(total, 2)` when you need to round.

This conversion ensures that the total price is displayed as an integer, which might be necessary for certain presentation or calculation requirements.

Chapter 3: Control Flow Statements

Conditional Statements

Conditional statements in C# allow you to execute different blocks of code based on whether a certain condition is true or false. The main conditional statements include:

- **if Statement:** Executes a block of code if a specified condition is true.

```
int num = 10;
if (num > 0)
{
    Console.WriteLine("Number is positive.");
}
```

- **else Statement:** Executes a block of code if the same condition is false.

```
int num = -5;
if (num > 0)
{
    Console.WriteLine("Number is positive.");
}
else
{
    Console.WriteLine("Number is not positive.");
}
```

- **else if Statement:** Allows you to specify a new condition to test if the previous condition is false.

```
int num = 0;
if (num > 0)
{
    Console.WriteLine("Number is positive.");
}
else if (num < 0)
{
    Console.WriteLine("Number is negative.");
}
else
{
    Console.WriteLine("Number is zero.");
}
```

Looping Statements

Looping statements in C# are used to execute a block of code repeatedly as long as a specified condition is true. Types of loops include:

- **for Loop:** Executes a block of code a specified number of times.

```
for (int i = 1; i <= 5; i++)
{
```

```
    Console.WriteLine(i);  
}
```

- **while Loop:** Executes a block of code as long as a specified condition is true.

```
int i = 1;  
while (i <= 5)  
{  
    Console.WriteLine(i);  
    i++;  
}
```

- **do-while Loop:** Similar to a while loop, but it always executes the block of code at least once before checking the condition.

```
int i = 1;  
do  
{  
    Console.WriteLine(i);  
    i++;  
}  
while (i <= 5);
```

Jump Statements

Jump statements in C# allow you to transfer control to another part of the program. Types of jump statements include:

- **break Statement:** Terminates the loop or switch statement and transfers control to the statement immediately following the loop or switch.

```
for (int i = 1; i <= 10; i++)  
{  
    if (i == 5)  
        break;  
    Console.WriteLine(i);  
}
```

- **continue Statement:** Skips the current iteration of a loop and continues with the next iteration.

```
for (int i = 1; i <= 5; i++)  
{  
    if (i == 3)  
        continue;  
    Console.WriteLine(i);  
}
```

- **foreach Loop:** Visits every item in a collection. It is the safest and most readable way to loop over lists and arrays.

```
string[] languages = ["C#", "JavaScript", "Python"];  
foreach (string language in languages)  
{
```

```

    Console.WriteLine(language);
}

```

- **switch Statement and switch Expression:** Choose between many values. The newer switch expression returns a value and supports patterns such as comparisons (<, >=) and and / or:

```

int score = 85;

string grade = score switch
{
    >= 90 => "A",
    >= 80 => "B",
    >= 70 => "C",
    _     => "Needs work"    // _ matches everything else
};
Console.WriteLine(grade);    // B

// Pattern matching with "is"
object value = 42;
if (value is int n and > 0)
{
    Console.WriteLine($"{n} is a positive integer.");
}

```

End-of-Chapter Exercises:

Multiple-Choice Questions:

1. Which loop is guaranteed to execute at least once?
 - A) for loop
 - B) while loop
 - C) do-while loop
 - D) foreach loop
2. What is the purpose of the break statement?
 - A) To skip the current iteration of a loop
 - B) To terminate a loop or switch statement
 - C) To transfer control to another part of the program
 - D) To check if a condition is true or false

Coding Tasks:

1. Write a C# program that finds the factorial of a number using a for loop.

```

Console.Write("Enter a number from 0 to 20: ");

if (!int.TryParse(Console.ReadLine(), out int number) || number is < 0 or > 20)
{
    Console.WriteLine("Please enter a whole number from 0 to 20.");
    return;
}

```

```
long factorial = 1;                // long holds larger values than int
for (int i = 1; i <= number; i++)
{
    factorial *= i;
}

Console.WriteLine($"Factorial of {number} is: {factorial}");
```

A long can hold factorials up to 20!. Beyond that the number overflows, which is why the program checks the input first.

Problem-Solving Scenarios:

1. Design a flowchart for a program that checks if a number is prime.

- **Flowchart Description:** A flowchart to check if a number is prime would typically involve starting with the number to be checked (n). You would then iterate through potential divisors from 2 up to the square root of n . For each divisor, check if n is divisible by the divisor without a remainder ($n \% \text{divisor} == 0$). If n is divisible by any number other than 1 and itself, then it is not prime. Otherwise, it is prime.

- **Example Flowchart:**

This flowchart illustrates the logical steps involved in determining whether a given number n is prime or not, based on the described algorithm.

Chapter 4: Methods and Functions

Method Declaration and Invocation

Methods in C# are blocks of code that perform a specific task. They are declared within a class and can be invoked (called) to execute the code inside them. A method declaration includes the method's access modifier, return type, name, parameters (if any), and body.

```
// Method declaration: access, return type, name, parameters
static int Add(int a, int b)
{
    return a + b;
}

// Method invocation
int sum = Add(5, 3);
Console.WriteLine($"Sum of 5 and 3 is: {sum}");

// Short methods can use an expression body
static int Multiply(int a, int b) => a * b;
Console.WriteLine(Multiply(4, 6)); // 24
```

Methods can also live inside a class, which is how larger programs are organized. A static method belongs to the class itself, so you can call it without creating an object:

```
public static class Calculator
{
    public static int Add(int a, int b) => a + b;
}

// In Program.cs, below the class or in another file:
// Console.WriteLine(Calculator.Add(5, 3));
```

Method Overloading

Method overloading in C# allows you to define multiple methods with the same name but different parameters. This enables you to use the same method name for different behaviors based on the types or number of parameters passed.

```
Printer.Display(10);
Printer.Display(3.5);
Printer.Display("Hello, world!");
Printer.Display("Hi there!", 3);

public static class Printer
{
    // Same method name, different parameter types
    public static void Display(int num) => Console.WriteLine($"Integer number: {num}");

    public static void Display(double num) => Console.WriteLine($"Double number: {num}");

    public static void Display(string message) => Console.WriteLine($"Message: {message}");
}
```

```
// Same method name, different number of parameters
public static void Display(string message, int times)
{
    for (int i = 0; i < times; i++)
    {
        Console.WriteLine($"Message {i + 1}: {message}");
    }
}
}
```

2026 Best Practice: Methods declared directly in top-level statements are local functions, and local functions cannot be overloaded. Put overloaded methods inside a class, as shown here.

Parameter Passing

Parameters in C# methods allow you to pass data into the method for processing. There are two ways to pass parameters: by value and by reference.

- **Passing Parameters by Value:** The default method of passing parameters in C#. A copy of the parameter's value is passed to the method. Changes made to the parameter inside the method do not affect the original value outside the method.
- **Passing Parameters by Reference:** Allows a method to modify the value of the parameter. The `ref` keyword is used to indicate that a parameter is passed by reference.

```
static void Increment(int num)
{
    num++;
    Console.WriteLine($"Inside Increment method: {num}");
}

static void IncrementByRef(ref int num)
{
    num++;
    Console.WriteLine($"Inside IncrementByRef method: {num}");
}

int number = 5;
Increment(number);                // passing by value
Console.WriteLine($"Outside method after Increment: {number}"); // 5

IncrementByRef(ref number);        // passing by reference
Console.WriteLine($"Outside method after IncrementByRef: {number}"); // 6
```

Two more parameter keywords are worth knowing. An out parameter lets a method hand back an extra value, which is how `int.TryParse` works. Optional and named arguments make calls easier to read:

```
static string Greet(string name, string greeting = "Hello") => $"{greeting}, {name}!";

Console.WriteLine(Greet("Ava"));                // Hello, Ava!
```

```

Console.WriteLine(Greet(greeting: "Welcome", name: "Ben")); // Welcome, Ben!

if (int.TryParse("123", out int parsed))
{
    Console.WriteLine(parsed + 1); // 124
}

```

Return Types

The return type specifies the type of data that a method can return to its caller. If a method does not return a value, its return type is void .

```

static int Add(int a, int b) => a + b;

static void DisplayMessage(string message) => Console.WriteLine(message);

// A method can return several values at once with a tuple
static (int Min, int Max) FindRange(int[] values) => (values.Min(), values.Max());

int sum = Add(5, 3);
DisplayMessage($"Sum is: {sum}");

var (min, max) = FindRange([4, 9, 2, 7]);
DisplayMessage($"Min: {min}, Max: {max}"); // Min: 2, Max: 9

```

End-of-Chapter Exercises:

Multiple-Choice Questions:

1. How do you declare a method in C# that does not return a value?

- A) void methodName() { }
 - B) int methodName() { }
 - C) double methodName() { }
 - D) string methodName() { }
2. What is method overloading?
- A) Defining multiple methods with the same name but different return types.
 - B) Defining multiple methods with the same name but different parameter types or number.
 - C) Defining multiple methods with different access modifiers.
 - D) Defining multiple methods within the same class.

Coding Tasks :

1. Write a method that takes two integers and returns their sum.

```

Console.WriteLine($"Sum of 10 and 5 is: {MathOperations.Sum(10, 5)}"); // 15

public static class MathOperations
{
    public static int Sum(int a, int b) => a + b;
}

```

Problem-Solving Scenarios:

1. Explain the difference between passing parameters by value and by reference with examples.

- **Passing Parameters by Value:** Copies the actual value of an argument into the parameter of the function. Changes made to the parameter inside the function have no effect on the argument.
- **Passing Parameters by Reference:** Passes a reference to the memory location of the actual argument, allowing the function to modify the parameter, which in turn modifies the argument.

```
int number = 5;

Increment(number);                // passing by value
Console.WriteLine($"Outside method after Increment: {number}");           // 5

IncrementByRef(ref number);       // passing by reference
Console.WriteLine($"Outside method after IncrementByRef: {number}");       // 6

static void Increment(int num)
{
    num++;
    Console.WriteLine($"Inside Increment method: {num}");
}

static void IncrementByRef(ref int num)
{
    num++;
    Console.WriteLine($"Inside IncrementByRef method: {num}");
}
```

In the example above, Increment method increments num locally, and changes do not affect number outside the method. IncrementByRef method, using ref keyword, modifies number directly, reflecting changes outside the method as well.

Chapter 5: Object-Oriented Programming (OOP) Basics

Classes and Objects

In C#, a class is a blueprint for creating objects. An object is an instance of a class that encapsulates data (fields) and behavior (methods). Classes define the structure and behavior of objects.

```
// Creating an object (instance) of the Person class
Person person1 = new() { Name = "John", Age = 30 };
person1.PrintDetails();    // Name: John, Age: 30

public class Person
{
    // Properties (auto-implemented) instead of public fields
    public required string Name { get; init; }
    public int Age { get; set; }

    // Method
    public void PrintDetails()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}");
    }
}
```

2026 Best Practice: The required keyword forces whoever creates the object to give a Name, and init means it can be set when the object is created but not changed afterward. Public **properties** are preferred over public **fields** because they let you add validation later without breaking other code.

Fields, Properties, and Methods

- **Fields:** Variables inside a class that hold data for each object.
- **Properties:** Accessors for private fields, allowing controlled access to data.
- **Methods:** Functions inside a class that define the behavior of objects.

A property can also run code when it is read or written. In C# 14 the `field` keyword gives you access to the hidden backing field, so you no longer have to declare one yourself:

```
public class Student
{
    public string Name
    {
        get;
        set => field = value.Trim();    // clean the value as it is stored
    } = "";

    public int Grade
}
```

```

    {
        get;
        set
        {
            ArgumentOutOfRangeException.ThrowIfNegative(value);
            field = value;
        }
    }
}

```

Constructors and Destructors

- **Constructors:** Special methods used to initialize objects. They have the same name as the class and can be overloaded.
- **Destructors (finalizers):** Rarely needed. The garbage collector frees memory for you automatically. If a class holds an unmanaged resource such as a file handle, implement `IDisposable` and use a `using` statement instead of writing a destructor.

```

var ava = new Person("Ava", 28);
ava.PrintDetails();

// A class with a primary constructor (C# 12): parameters go right after the class name
public class Person(string name, int age)
{
    public string Name { get; } = name;
    public int Age { get; } = age;

    public void PrintDetails() => Console.WriteLine($"Name: {Name}, Age: {Age}");
}

```

For classes that mainly hold data, use a **record**. Records give you value-based equality, a readable `ToString()` and the `with` keyword for making modified copies:

```

var a = new Person("Ava", 28);
var b = a with { Age = 29 }; // a copy with one change
Console.WriteLine(b);       // Person { Name = Ava, Age = 29 }
Console.WriteLine(a == new Person("Ava", 28)); // True

public record Person(string Name, int Age);

```

A class that owns a resource implements `IDisposable`:

```

using var logger = new Logger(); // Dispose is called automatically at the end of the scope
logger.Log("Started");

public sealed class Logger : IDisposable
{
    private readonly StreamWriter _writer = new("log.txt");

    public void Log(string message) => _writer.WriteLine(message);

    public void Dispose() => _writer.Dispose();
}

```

Access Modifiers

Access modifiers control the visibility and accessibility of classes, fields, properties, and methods within a program.

- **Public:** Accessible from any other class or assembly.
- **Private:** Accessible only from within the same class.
- **Protected:** Accessible from within the same class or derived classes.
- **Internal:** Accessible only within the same assembly.
- **Protected Internal:** Accessible within the same assembly or from a derived class in another assembly.

Inheritance and Polymorphism

- **Inheritance:** Mechanism by which one class (derived class) inherits properties and behavior from another class (base class).
- **Polymorphism:** Ability of objects to take on multiple forms. Achieved through method overriding and method overloading.

```
Person p = new Student { Name = "Ava", Age = 20, StudentId = 1001 };
p.PrintDetails(); // calls the Student version

public class Person
{
    public required string Name { get; init; }
    public int Age { get; init; }

    // virtual: a derived class may replace this behavior
    public virtual void PrintDetails() => Console.WriteLine($"Name: {Name}, Age: {Age}");
}

public class Student : Person
{
    public int StudentId { get; init; }

    public void Study() => Console.WriteLine($"{Name} is studying.");

    // override: polymorphism in action
    public override void PrintDetails() =>
        Console.WriteLine($"Student {Name} (ID {StudentId}), age {Age}");
}
```

Use **interfaces** to describe what a type can do without saying how. Classes can implement many interfaces but inherit from only one class. Prefer composition ("has a") over deep inheritance ("is a") when you can:

```
IShape[] shapes = [new Circle(2), new Rectangle(3, 4)];
foreach (IShape shape in shapes)
{
    Console.WriteLine($"{shape.GetType().Name}: {shape.Area():F2}");
}
```

```

public interface IShape
{
    double Area();
}

public class Circle(double radius) : IShape
{
    public double Area() => Math.PI * radius * radius;
}

public class Rectangle(double width, double height) : IShape
{
    public double Area() => width * height;
}

```

End-of-Chapter Exercises:

Multiple-Choice Questions:

1. What is the purpose of a constructor in C#?
 - A) To destroy an object
 - B) To initialize an object
 - C) To define inheritance
 - D) To define polymorphism
2. Which keyword is used to inherit a class in C#?
 - A) new
 - B) this
 - C) base
 - D) :

Coding Tasks:

1. Create a class Person with properties for Name and Age , and a method that prints the person's details.

```

// Create an object of the Person class
Person person1 = new() { Name = "Alice", Age = 25 };

// Call the method to print details
person1.PrintDetails();

public class Person
{
    public required string Name { get; init; }
    public int Age { get; set; }

    public void PrintDetails() => Console.WriteLine($"Name: {Name}, Age: {Age}");
}

```

Problem-Solving Scenarios:

1. Design a simple inheritance hierarchy for a school management system.

- **Inheritance Hierarchy Example:**

```
Student student = new() { Name = "Ava", Age = 17, StudentId = 1001, GradeLevel = 11 };
Teacher teacher = new() { Name = "Mr. Lee", Age = 40, SubjectTaught = "Math" };

student.Study();    // Ava is studying.
teacher.Teach();    // Mr. Lee is teaching Math.

public class Person
{
    public required string Name { get; init; }
    public int Age { get; init; }

    public void PrintDetails() => Console.WriteLine($"Name: {Name}, Age: {Age}");
}

public class Student : Person
{
    public int StudentId { get; init; }
    public int GradeLevel { get; init; }

    public void Study() => Console.WriteLine($"{Name} is studying.");
}

public class Teacher : Person
{
    public required string SubjectTaught { get; init; }

    public void Teach() => Console.WriteLine($"{Name} is teaching {SubjectTaught}.");
}
```

In this example, Student and Teacher classes inherit from the Person class, inheriting the Name and Age properties and the PrintDetails() method. Additionally, Student and Teacher have their specific properties (StudentId , GradeLevel for Student ; SubjectTaught for Teacher) and methods (Study() for Student ; Teach() for Teacher).

Chapter 6: Arrays and Collections

Single-Dimensional and Multi-Dimensional Arrays

Arrays in C# are used to store multiple values of the same data type sequentially. They can be single-dimensional or multi-dimensional.

- **Single-Dimensional Array:**

```
int[] numbers = new int[5]; // Declaration and initialization
numbers[0] = 10;
numbers[1] = 20;
// Accessing elements
Console.WriteLine(numbers[0]); // Output: 10
```

- **Multi-Dimensional Array:**

```
int[,] matrix = new int[2, 3]; // 2 rows, 3 columns
matrix[0, 0] = 1;
matrix[0, 1] = 2;
// Accessing elements
Console.WriteLine(matrix[0, 1]); // Output: 2
```

Modern C# lets you create arrays with a **collection expression** (square brackets). Use `^1` for "last item", ranges such as `1..3` to take a slice, and the spread operator `..` to combine collections:

```
int[] scores = [90, 85, 77, 92];
Console.WriteLine(scores[^1]); // 92 (last item)
int[] firstTwo = scores[..2]; // [90, 85]
int[] all = [..scores, 100]; // [90, 85, 77, 92, 100]
Console.WriteLine(scores.Length); // 4
```

Collections in C#

Collections are classes in C# that are used to store and manage groups of related objects. They provide more flexibility and functionality compared to arrays.

- **List<T>:** Dynamic array that stores elements of a specified type.
- **Dictionary<TKey, TValue>:** Collection of key-value pairs where each key is unique.
- **Queue<T>:** FIFO (First-In-First-Out) collection.
- **Stack<T>:** LIFO (Last-In-First-Out) collection.

```
// List example
List<int> numbers = [];
numbers.Add(10);
numbers.Add(20);
Console.WriteLine(numbers[0]); // Output: 10

// Dictionary example
Dictionary<string, int> ages = new()
```

```

{
    ["Alice"] = 25,
    ["Bob"] = 30,
};
Console.WriteLine(ages["Bob"]);           // Output: 30

// Safe lookup: no exception if the key is missing
if (ages.TryGetValue("Carol", out int carolAge))
{
    Console.WriteLine(carolAge);
}
else
{
    Console.WriteLine("Carol was not found.");
}

```

Basic Operations on Collections

Collections support various operations such as adding, removing, accessing elements, iterating, and sorting.

- **Adding Elements:**

```

List<int> numbers = new List<int>();
numbers.Add(10);

```

- **Removing Elements:**

```

numbers.Remove(10);

```

- **Accessing Elements:**

```

Console.WriteLine(numbers[0]);

```

- **Iterating through Collection:**

```

foreach (int num in numbers)
{
    Console.WriteLine(num);
}

```

```

List<string> names = ["Charlie", "alice", "Bob"];
names.Sort(StringComparer.OrdinalIgnoreCase); // alice, Bob, Charlie
Console.WriteLine(names.Contains("Bob"));      // True
Console.WriteLine(names.Count);                // 3

HashSet<int> unique = [1, 2, 2, 3, 3, 3];      // duplicates are ignored
Console.WriteLine(unique.Count);               // 3

```

Choose the right collection: `List<T>` for ordered items, `Dictionary<TKey, TValue>` for lookups by key, `HashSet<T>` for unique values, `Queue<T>` and `Stack<T>` for first-in and last-in processing.

End-of-Chapter Exercises:

Multiple-Choice Questions:

1. How do you declare a single-dimensional array in C#?

- A) `int[] numbers = new int[];`
 - B) `int[] numbers = new int[5];`
 - C) `array numbers = new array(5);`
 - D) `array numbers = new int[5];`
2. Which collection allows for key-value pairs?
- A) `List<T>`
 - B) `Array`
 - C) `Dictionary<TKey, TValue>`
 - D) `Stack<T>`

Coding Tasks:

1. Write a program to sort an array of integers.

```
int[] numbers = [4, 2, 7, 1, 5];

Array.Sort(numbers); // sorts the array in place
Console.WriteLine(string.Join(" ", numbers)); // 1 2 4 5 7
```

Problem-Solving Scenarios:

1. Explain a scenario where a `Dictionary<TKey, TValue>` would be more appropriate than a `List<T>` .

- **Scenario:** Managing a phone book where each contact has a unique name and associated phone number.
- **Explanation:** In this scenario, using a `Dictionary<string, string>` (where `string` represents name and phone number) would be more appropriate than a `List<string>` . This is because a `Dictionary<TKey, TValue>` allows efficient lookup and retrieval of phone numbers by name (key-value pair), leveraging the uniqueness of keys (names) compared to a `List<string>` which would not provide efficient lookup operations.

Chapter 7: Exception Handling

Try, Catch, Finally Blocks

Exception handling in C# allows you to manage runtime errors gracefully. Key components include try , catch , and finally blocks:

- **Try Block:** Contains code that may cause an exception.
- **Catch Block:** Handles exceptions that occur within the try block.
- **Finally Block:** Executes code after try block, whether an exception is thrown or not (optional).

```
int numerator = 10;
int denominator = 0;

try
{
    int result = numerator / denominator;    // throws DivideByZeroException
    Console.WriteLine($"Result: {result}");
}
catch (DivideByZeroException ex)
{
    Console.WriteLine($"Error: Cannot divide by zero. ({ex.Message})");
}
finally
{
    Console.WriteLine("Finally block executed.");
}
```

2026 Best Practice: Catch the most specific exception you can. Avoid an empty catch and avoid catching the base `Exception` except at the very top of your program. Do not use exceptions for normal decisions; if you can check a condition first (for example with `TryParse`), do that instead.

Throwing Exceptions

Exceptions can be explicitly thrown using the `throw` keyword to indicate abnormal conditions in code execution.

```
static void ValidateAge(int age)
{
    if (age < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(age), "Age cannot be negative.");
    }

    Console.WriteLine($"Valid age: {age}");
}

try
```

```

{
    ValidateAge(-5);    // throws ArgumentOutOfRangeException
}
catch (ArgumentOutOfRangeException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}

```

.NET also provides short guard helpers, such as `ArgumentOutOfRangeException.ThrowIfNegative(age);`, `ArgumentNullException.ThrowIfNull(value);` and `ArgumentException.ThrowIfNullOrWhiteSpace(text);`. If you catch an exception and need to pass it along, write `throw;` (not `throw ex;`) so the original stack trace is kept.

Custom Exceptions

Custom exceptions allow you to create specific exception types tailored to your application's needs by deriving from `Exception` class.

```

static void ValidateInput(string input)
{
    if (input.Length < 5)
    {
        throw new InvalidUserInputException("Input length must be at least 5 characters.");
    }

    Console.WriteLine($"Valid input: {input}");
}

try
{
    ValidateInput("Hi");    // throws InvalidUserInputException
}
catch (InvalidUserInputException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}

public class InvalidUserInputException : Exception
{
    public InvalidUserInputException() { }

    public InvalidUserInputException(string message) : base(message) { }

    public InvalidUserInputException(string message, Exception innerException)
        : base(message, innerException) { }
}

```

Name custom exceptions with the word `Exception` at the end and provide the three standard constructors shown above.

End-of-Chapter Exercises:

Multiple-Choice Questions:

1. What is the purpose of the finally block?

- A) To handle exceptions
 - B) To throw exceptions
 - C) To execute cleanup code
 - D) To declare custom exceptions
2. How do you throw an exception in C#?
- A) catch (Exception ex) { }
 - B) throw new Exception("Message");
 - C) try { }
 - D) finally { }

Coding Tasks:

1. Write a program that handles divide-by-zero exceptions.

```
int numerator = 10;
int denominator = 0;

try
{
    int result = numerator / denominator;    // throws DivideByZeroException
    Console.WriteLine($"Result: {result}");
}
catch (DivideByZeroException)
{
    Console.WriteLine("Error: Cannot divide by zero.");
}
```

Problem-Solving Scenarios:

1. Design a custom exception class for handling invalid user input.

```
try
{
    ValidateInput("Hi");    // throws InvalidUserInputException
}
catch (InvalidUserInputException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}

static void ValidateInput(string input)
{
    if (input.Length < 5)
    {
        throw new InvalidUserInputException("Input length must be at least 5 characters.");
    }

    Console.WriteLine($"Valid input: {input}");
}
```

```
}  
  
public class InvalidUserInputException(string message) : Exception(message);
```

In this chapter, you've learned how to handle exceptions using try , catch , and finally blocks, throw exceptions explicitly, and create custom exceptions to handle specific scenarios in your C# programs effectively

Chapter 8: File I/O and Streams

Reading from and Writing to Files

File I/O (Input/Output) operations in C# involve reading from and writing to files using `StreamReader` and `StreamWriter` classes.

- **Reading from a File:**

```
string filePath = "sample.txt";

try
{
    // File.ReadLines reads one line at a time and closes the file for you
    foreach (string line in File.ReadLines(filePath))
    {
        Console.WriteLine(line);
    }
}
catch (FileNotFoundException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
```

If you need more control, a `StreamReader` with a `using` declaration closes the file automatically at the end of the scope:

```
using var reader = new StreamReader("sample.txt");
string? line;
while ((line = reader.ReadLine()) is not null)
{
    Console.WriteLine(line);
}
```

- **Writing to a File:**

```
string filePath = "output.txt";

try
{
    string[] lines = ["Hello, World!", "This is a sample text."];
    File.WriteAllLines(filePath, lines);
    // Use File.AppendAllText(filePath, "more text") to add to an existing file
}
catch (IOException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
```

For anything that could take time, such as large files or network drives, use the **async** versions so your program stays responsive:

```
await File.WriteAllTextAsync("notes.txt", "Remember to practice C# today.");
string text = await File.ReadAllTextAsync("notes.txt");
Console.WriteLine(text);
```

Working with Streams

Streams in C# provide a uniform way to access input and output data. They are used for reading or writing bytes of data.

- **Reading from a Stream:**

```
string filePath = "sample.txt";

try
{
    using var fileStream = new FileStream(filePath, FileMode.Open, FileAccess.Read);

    byte[] buffer = new byte[1024];
    int bytesRead;
    while ((bytesRead = fileStream.Read(buffer, 0, buffer.Length)) > 0)
    {
        Console.WriteLine($"Read {bytesRead} bytes.");
    }
}
catch (IOException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
```

- **Writing to a Stream:**

```
string filePath = "output.txt";

try
{
    using var fileStream = new FileStream(filePath, FileMode.Create, FileAccess.Write);

    byte[] data = System.Text.Encoding.UTF8.GetBytes("Hello, World!");
    fileStream.Write(data, 0, data.Length);
}
catch (IOException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
```

Handling File Exceptions

File-related operations can throw exceptions such as `FileNotFoundException`, `IOException`, etc. It's essential to handle these exceptions to prevent application crashes and provide meaningful error messages to users.

```
string filePath = Path.Combine("data", "sample.txt"); // builds a path that works on every OS

try
{
```

```

        using var reader = new StreamReader(filePath);
        Console.WriteLine(reader.ReadToEnd());
    }
    catch (FileNotFoundException ex)
    {
        Console.WriteLine($"File not found: {ex.Message}");
    }
    catch (DirectoryNotFoundException ex)
    {
        Console.WriteLine($"Folder not found: {ex.Message}");
    }
    catch (IOException ex)
    {
        Console.WriteLine($"IO Error: {ex.Message}");
    }
}

```

List the most specific exception first. `FileNotFoundException` and `DirectoryNotFoundException` both derive from `IOException`, so `IOException` must come last. Never build file paths from raw user input without checking them, to avoid reading or overwriting files you did not intend.

End-of-Chapter Exercises:

Multiple-Choice Questions:

- Which class is used for reading text from a file?
 - A) `FileStream`
 - B) `StreamReader`
 - C) `StreamWriter`
 - D) `FileReader`
- How do you handle file not found exceptions?
 - A) Use try-catch block with `FileNotFoundException`
 - B) Use try-catch block with `IOException`
 - C) Use try-catch block with `FileNotFound`
 - D) Use try-catch block with `FileError`

Coding Tasks:

- Write a program to read a text file and print its content to the console.

```

try
{
    using var reader = new StreamReader("sample.txt");

    string? line;
    while ((line = reader.ReadLine()) is not null)
    {
        Console.WriteLine(line);
    }
}
catch (FileNotFoundException ex)
{
}

```

```
        Console.WriteLine($"File not found: {ex.Message}");  
    }  
    catch (IOException ex)  
    {  
        Console.WriteLine($"IO Error: {ex.Message}");  
    }
```

Problem-Solving Scenarios:

1. Explain how you would handle large files efficiently in C#.

- **Handling Large Files Efficiently:**
- Use buffered reading and writing (StreamReader , StreamWriter) to minimize disk access.
- Process files in chunks rather than loading entire file into memory.
- Utilize asynchronous I/O operations (ReadAsync() , WriteAsync()) to improve responsiveness.
- Implement proper error handling and recovery mechanisms for uninterrupted file processing.

Chapter 9: Basic Understanding of LINQ

Introduction to LINQ

LINQ (Language Integrated Query) is a feature in C# that allows for querying data in a type-safe manner directly from .NET objects, XML, SQL databases, and other data sources.

- **Benefits of LINQ:**
- Provides a unified syntax for querying different data sources.
- Type safety at compile-time.
- Enhances readability and maintainability of code.

Basic LINQ Queries

LINQ queries are written in a declarative syntax using keywords like `from` , `where` , `select` , `orderby` , `group by` , etc.

- **Example: Filtering and Selecting Data**

```
List<int> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Query syntax: LINQ query to find all even numbers
var evenNumbers = from num in numbers
                  where num % 2 == 0
                  select num;

foreach (var num in evenNumbers)
{
    Console.WriteLine(num);    // Output: 2, 4, 6, 8, 10
}

// Method syntax: the same query written with lambda expressions
var evens = numbers.Where(n => n % 2 == 0).ToList();
var total = numbers.Sum();           // 55
var biggestThree = numbers.OrderByDescending().Take(3);    // 10, 9, 8
```

Method syntax (`Where`, `Select`, `OrderBy`) is the most common style in professional code. A LINQ query does not run until you loop over it or call `ToList()`, `ToArray()`, `Count()` or similar. This is called **deferred execution**.

Using LINQ with Collections

LINQ can be used with various types of collections (`List<T>` , `Array` , `Dictionary<TKey, TValue>` , etc.) to perform operations such as filtering, sorting, grouping, and projecting data.

- **Example: Using LINQ with List**

```

List<string> fruits = ["apple", "banana", "cherry", "date"];

// LINQ query to filter fruits starting with 'a'
var filteredFruits = from fruit in fruits
                    where fruit.StartsWith('a')
                    select fruit;

foreach (var fruit in filteredFruits)
{
    Console.WriteLine(fruit);    // Output: apple
}

// Grouping and projecting with method syntax
var byLength = fruits
    .GroupBy(f => f.Length)
    .Select(g => $"{g.Key} letters: {string.Join(", ", g)}");

foreach (var line in byLength)
{
    Console.WriteLine(line);    // 5 letters: apple / 6 letters: banana, cherry / 4 letters: date
}

```

End-of-Chapter Exercises:

Multiple-Choice Questions:

1. What does LINQ stand for?

- A) Language-Integrated Query
 - B) Linked-Integrated Query
 - C) Library-Integrated Query
 - D) Logic-Integrated Query
2. Which method is used to filter a collection in LINQ?
- A) Filter()
 - B) Find()
 - C) Where()
 - D) Select()

Coding Tasks:

1. Write a LINQ query to find all even numbers in a list.

```

List<int> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var evenNumbers = from num in numbers
                 where num % 2 == 0
                 select num;

foreach (var num in evenNumbers)
{
    Console.WriteLine(num);    // Output: 2, 4, 6, 8, 10
}

```

Problem-Solving Scenarios:

1. Explain how LINQ can simplify querying collections compared to traditional loops.

- **Simplification with LINQ:**
- **Readability:** LINQ queries use a declarative syntax that is more readable and closer to natural language.
- **Ease of Use:** Eliminates the need for nested loops and manual iteration over collections.
- **Type Safety:** Provides compile-time type checking, reducing runtime errors.
- **Code Reusability:** LINQ queries can be reused across different data sources without modification.
- **Example Comparison:**

```
List<int> numbers = [1, 2, 3, 4, 5];

// Traditional approach with loops
List<int> evenWithLoop = [];
foreach (int num in numbers)
{
    if (num % 2 == 0)
    {
        evenWithLoop.Add(num);
    }
}

// Using LINQ
var evenWithLinq = numbers.Where(num => num % 2 == 0).ToList();
```

In this comparison, LINQ simplifies the code by combining filtering and selecting logic into a single readable statement, enhancing code maintainability and reducing the potential for bugs.

Chapter 10: Basic Concepts of Debugging and Testing

Debugging Techniques

Debugging is the process of identifying and resolving errors or bugs in software applications. Effective debugging techniques involve using tools and methods to locate and fix issues.

- **Common Debugging Techniques:**
- **Using Breakpoints:** Pauses code execution at specific points to examine program state.
- **Inspecting Variables:** Checking the values of variables to understand their state.
- **Logging:** Adding log messages to track program flow and variable values.
- **Stepping Through Code:** Executing code line-by-line to trace execution paths.
- **Debugging Tools:** Utilizing integrated development environment (IDE) tools like Visual Studio debugger.

Writing Unit Tests

Unit testing is a software testing method where individual units or components of a program are tested to ensure they function as expected. In C#, unit tests are commonly written using frameworks like NUnit, MSTest, or xUnit.

- **Benefits of Unit Testing:**
- Validates correctness of code behavior.
- Facilitates early bug detection and regression testing.
- Improves code quality and maintainability.
- Supports refactoring and code changes with confidence.

A first unit test with xUnit. Create a test project next to your app (`dotnet new xunit -n MyApp.Tests`), add a reference to your app, and run all tests with `dotnet test`:

```
// In your app
public static class Calculator
{
    public static int Add(int a, int b) => a + b;
}
```

```
// In MyApp.Tests
using Xunit;

public class CalculatorTests
{
    [Fact]
    public void Add_FiveAndThree_ReturnsEight()
    {
```

```

        int result = Calculator.Add(5, 3);    // Arrange and Act

        Assert.Equal(8, result);              // Assert
    }

    [Theory]
    [InlineData(1, 2, 3)]
    [InlineData(-1, 1, 0)]
    [InlineData(0, 0, 0)]
    public void Add_ManyInputs_ReturnsSum(int a, int b, int expected)
    {
        Assert.Equal(expected, Calculator.Add(a, b));
    }
}

```

Name tests after what they check, and follow the **Arrange, Act, Assert** pattern. For logging in real applications, use `ILogger` from `Microsoft.Extensions.Logging` instead of `Console.WriteLine`, so you can control log levels and send logs to files or cloud services.

End-of-Chapter Exercises:

Multiple-Choice Questions:

1. What is the purpose of a breakpoint in debugging?

- A) To terminate program execution
 - B) To pause program execution at a specific point
 - C) To skip over code sections
 - D) To display error messages
2. Which framework is commonly used for unit testing in C#?
- A) JUnit
 - B) NUnit
 - C) Mockito
 - D) PyUnit

Coding Tasks:

1. Write a simple unit test for a method that adds two numbers.

```

// Calculator.cs (your app)
public static class Calculator
{
    public static int Add(int a, int b) => a + b;
}

// CalculatorTests.cs (your test project, using xUnit)
using Xunit;

public class CalculatorTests
{
    [Fact]

```

```
public void Add_ShouldReturnSum()
{
    // Arrange
    int a = 3, b = 5;

    // Act
    int result = Calculator.Add(a, b);

    // Assert
    Assert.Equal(8, result);
}
```

If your team uses NUnit instead, mark the class with `[TestFixture]`, the method with `[Test]` and write `Assert.That(result, Is.EqualTo(8));`.

Problem-Solving Scenarios:

1. Explain a scenario where debugging helped you find and fix a critical bug.

Scenario Explanation:

- **Issue:** In a web application, users reported that they couldn't submit a form.
- **Debugging Process:**
- **Step 1:** Added breakpoints in form submission handler.
- **Step 2:** Stepped through code to track data flow and variable values.
- **Step 3:** Identified a validation logic error preventing form submission.
- **Step 4:** Fixed the validation condition and tested the fix.
- **Outcome:** Users were able to submit forms successfully after the bug fix, demonstrating the effectiveness of debugging in resolving critical issues.

In this chapter, you've learned fundamental debugging techniques and the importance of writing unit tests to ensure software reliability and maintainability. These practices are essential for producing high-quality code and identifying and fixing bugs efficiently during development.

Chapter 11: Introduction to Visual Studio

Overview of the Visual Studio IDE

Visual Studio is an integrated development environment (IDE) used for developing software applications across multiple platforms. It supports various programming languages, including C#, and provides tools for coding, debugging, and deploying applications.

- **Key Features of Visual Studio:**
- **Code Editor:** Supports syntax highlighting, code completion, and refactoring.
- **Solution Explorer:** Organizes project files and folders.
- **Tool Windows:** Includes Solution Explorer, Properties, Output, Error List, and more.
- **Integrated Debugger:** Facilitates debugging with breakpoints, watch windows, call stack, and immediate window.
- **NuGet Package Manager:** Manages third-party libraries and dependencies.
- **Version Control Integration:** Supports Git and GitHub, and other version control systems.

Creating, Building, and Running a C# Project

Visual Studio simplifies the process of creating, building, and running C# projects.

- **Steps to Create a New C# Project:**
 1. Open Visual Studio.
 2. Go to File > New > Project... .
 3. Select **Console App** (the C# template that targets .NET 10).
 4. Provide a name and location for the project.
 5. Click Create to generate the project structure.
- **Building and Running a C# Project:**
 - Build: Press Ctrl + Shift + B or go to Build > Build Solution .
 - Run: Press F5 or go to Debug > Start Debugging (or Start Without Debugging).

Using the Debugger

The Visual Studio debugger helps identify and fix issues in code during development.

- **Debugger Features:**
- **Breakpoints:** Pause execution at specific lines to inspect variables and program state.
- **Watch Windows:** Monitor variable values and expressions.
- **Call Stack:** View the sequence of method calls leading to the current point in execution.
- **Immediate Window:** Execute code and evaluate expressions during debugging.

End-of-Chapter Exercises:

Multiple-Choice Questions:

1. How do you create a new project in Visual Studio?

- A) File > New > Solution
 - B) File > New > Project
 - C) File > Open > Project
 - D) File > Create > Project
2. Which window is used to view the call stack during debugging?
- A) Solution Explorer
 - B) Properties
 - C) Call Stack
 - D) Output

Coding Tasks:

1. Create a new console application in Visual Studio and write a simple "Hello, World!" program.

• **Steps:**

1. Open Visual Studio.
2. Go to File > New > Project... .
3. Select **Console App** (the C# template that targets .NET 10).
4. Name your project and click Create .
5. Replace the generated code in Program.cs with the following:

```
Console.WriteLine("Hello, World!");
```

6. Press F5 to build and run the program.

Problem-Solving Scenarios:

1. Explain how you would use Visual Studio to debug a complex application.

• **Debugging Complex Applications:**

- **Setting Breakpoints:** Identify critical points in the code where you suspect issues.
- **Inspecting Variables:** Use watch windows and immediate window to monitor variable values.
- **Analyzing Call Stack:** Trace the sequence of method calls to understand program flow.
- **Handling Exceptions:** Use exception handling and debugger to catch and diagnose runtime errors.
- **Performance Profiling:** Use profiling tools to analyze and optimize application performance.
- **Example Scenario:** Debugging a web application:
- **Issue:** Users report intermittent errors during checkout.
- **Debugging Steps:**
 - Set breakpoints in checkout-related methods.
 - Monitor session variables and database interactions.
 - Analyze HTTP requests and responses for anomalies.
 - Use logging and tracing to capture detailed information.
 - Collaborate with team members using version control and remote debugging tools.

In this chapter, you've explored the fundamentals of Visual Studio IDE, including project creation, debugging techniques, and the essential tools for software development. Visual Studio enhances productivity by providing robust features for coding, debugging, and deploying applications efficiently.

Modern C# for 2026

This chapter collects the modern features used throughout the book and shows how a professional C# project is set up today.

Versions at a Glance

.NET is the platform that runs C# programs and C# is the language. A new .NET version arrives every November. Even-numbered releases are **Long-Term Support (LTS)**, and this book targets **.NET 10 (LTS) with C# 14** and **Visual Studio 2026**. Check dotnet.microsoft.com for the newest supported version before you start a project.

A Modern Project File

When you run `dotnet new console`, you get a small project file like this. Leave nullable reference types and implicit usings turned on:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

What's New in C# 14

Two C# 14 features you may see in new code are the `field` keyword in properties (shown in Chapter 5) and **extension members**, which let you add properties and methods to types you do not own:

```
Console.WriteLine(" ".IsBlank);           // True
Console.WriteLine("hello".Shout());        // HELLO!

public static class StringExtensions
{
    extension(string text)
    {
        public bool IsBlank => string.IsNullOrEmpty(text);

        public string Shout() => text.ToUpperInvariant() + "!";
    }
}
```

C# 14 also allows null-conditional assignment, such as `customer?.Name = "Ava";`, which assigns only if `customer` is not null.

Async and Await

Use `async` and `await` for work that waits on something slow, like a web request. Top-level programs can use `await` directly:

```
using var client = new HttpClient();
client.Timeout = TimeSpan.FromSeconds(10);

try
{
    string html = await client.GetStringAsync("https://example.com");
    Console.WriteLine($"Downloaded {html.Length} characters.");
}
catch (HttpRequestException ex)
{
    Console.WriteLine($"Request failed: {ex.Message}");
}
```

Name methods that return a `Task` with the suffix `Async`, for example `LoadUsersAsync()`.

A Tiny Web API

With the `dotnet new web` template, you can build a web API in four lines using ASP.NET Core minimal APIs:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello from BioGlytch Studios!");
app.MapGet("/greet/{name}", (string name) => $"Hello, {name}!");

app.Run();
```

2026 C# Best Practices

- **Use a current, supported .NET.** Target the latest LTS release and keep the SDK updated for security fixes.
- **Follow naming conventions.** PascalCase for types, methods and properties; camelCase for parameters and local variables; `_camelCase` for private fields; an `I` prefix for interfaces (`IShape`); the `Async` suffix for async methods.
- **Keep nullable reference types on** and treat warnings seriously. Consider `<TreatWarningsAsErrors>true</TreatWarningsAsErrors>` for your own projects.
- **Prefer properties over public fields**, and make data `init` or read-only whenever it should not change.
- **Use records** for data and **interfaces** for behavior. Prefer composition over deep inheritance, and mark classes `sealed` when they are not designed for inheritance.
- **Use `var` when the type is obvious**, and write the type when it helps readers.
- **Validate input.** Use `TryParse`, check ranges, and use guard methods like `ArgumentNullException.ThrowIfNull`.
- **Handle exceptions precisely.** Catch specific exceptions, never swallow them silently, and use `throw;` to rethrow.
- **Clean up resources with `using`.** Use `using var` for files, streams, database connections and `HttpClient` handlers.
- **Use `async` and `await` for I/O**, never `.Result` or `.Wait()`, and pass a `CancellationToken` for long operations.
- **Prefer LINQ and collection expressions** for clear, short code, but avoid LINQ chains that are hard to read.
- **Protect secrets.** Never put passwords, API keys or connection strings in source code. Use environment variables, user secrets (`dotnet user-secrets`) or a secret manager. Use parameterized queries for databases to prevent SQL injection.
- **Test and automate.** Write unit tests with `xUnit`, run `dotnet format` and analyzers, and use Git with a continuous-integration build.
- **Write small methods with one job**, meaningful names and comments that explain *why*, not *what*.

Appendix

Appendix A: C# Programming Environment Setup

Installing Visual Studio

1. **Download Visual Studio** : Go to the Visual Studio website and download the free Community edition of Visual Studio 2026. (You can also use Visual Studio Code with the C# Dev Kit extension, or JetBrains Rider.)
2. **Install Visual Studio** : Run the installer and select the workloads you need, such as .NET desktop development or ASP.NET and web development.
3. **Create a New Project** : Open Visual Studio, go to File > New > Project, select Console App (C#, targeting .NET 10), and click Create. If you prefer the command line, install the .NET SDK and run `dotnet new console -n MyFirstApp`.

Setting Up a C# Project

1. Create a Console Application :

- Open Visual Studio.
- Select Create a new project .
- Choose Console App and click Next .
- Name your project and click Create .

2. Write Your First Program :

- Open Program.cs .
- Replace the code with:

```
Console.WriteLine("Hello, World!");
```

- Press F5 to run the program.

Appendix B: Key C# Syntax and Commands

Data Types and Variables

- Declaration: `int number;`
- Initialization: `int number = 5;`
- Primitive Types: `int` , `double` , `char` , `bool` , `string`

Control Flow Statements

- If Statement:

```
if (condition)
{
    // code to execute
}
```

- For Loop:

```
for (int i = 0; i < 10; i++)
{
    // code to execute
}
```

- While Loop:

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

Methods and Functions

- Declaration:

```
void MyMethod()
{
    // code to execute
}
```

- Invocation:

```
MyMethod();
```

Classes and Objects

- Class Definition:

```
public class Person
{
    public required string Name { get; init; }
    public int Age { get; set; }
}
```

- Object Creation:

```
Person person = new() { Name = "Ava", Age = 28 };
```

Appendix C: LINQ Examples

Basic LINQ Query

```
List<int> numbers = [1, 2, 3, 4, 5];
var evenNumbers = from num in numbers
                  where num % 2 == 0
                  select num;
```

LINQ with Methods

```
var evenNumbers = numbers.Where(num => num % 2 == 0).ToList();  
var squares = numbers.Select(n => n * n).ToList();  
var firstBig = numbers.FirstOrDefault(n => n > 3);
```

Glossary

Array : A data structure that contains a collection of elements of the same type.

Class : A blueprint for creating objects, defining their data and behavior.

Constructor : A special method in a class used to initialize objects.

Debugging : The process of identifying and fixing errors in code.

Exception Handling : Mechanisms to handle runtime errors using try, catch, and finally blocks.

IDE (Integrated Development Environment) : A software application that provides comprehensive facilities to programmers for software development.

Inheritance : A mechanism by which one class can inherit properties and methods from another class.

LINQ (Language Integrated Query) : A set of features that adds query capabilities to the .NET languages.

Method : A block of code that performs a specific task.

Async/Await : Keywords that let a program wait for slow work, such as a file or network request, without freezing.

Collection Expression : A short way to create arrays and lists using square brackets, such as [1, 2, 3].

NuGet : The package manager that lets you add libraries to a .NET project.

Nullable Reference Type : A feature that lets the compiler warn you when a value that might be null is used unsafely; written with a question mark, as in `string?`.

Object : An instance of a class.

Pattern Matching : Testing a value against a shape or condition, using `is` or a switch expression.

Overloading : Defining multiple methods with the same name but different parameters.

Parameter : A variable used in a method to refer to one of the pieces of data provided as input to the method.

Polymorphism : The ability of different classes to be treated as instances of the same class through inheritance.

Record : A short way to declare a class (or struct) that mainly holds data and compares by value.

Stream : An abstract representation of a sequence of bytes, such as data from a file or a network connection.

Top-Level Statements : A C# 9 feature that lets a program start without an explicit class and Main method.

Unit Test : A small automated test that checks one piece of code, usually written with xUnit, NUnit or MSTest.

Variable : A storage location identified by a memory address and a symbolic name, used to store data.

Visual Studio : An integrated development environment (IDE) from Microsoft used for developing applications.

This appendix and glossary provide additional resources and definitions to support your understanding of the concepts covered in the book.