

Published by Randy Fadler

July 2025

# Learning C#

This booklet aims to provide a comprehensive yet accessible introduction to C# programming for beginners. The focus will be on core concepts, building a strong foundation, and equipping the reader to confidently explore more advanced topics.

## 1. Introduction to C#

* What is C# and its place in the .NET ecosystem? C# is a modern, object-oriented language developed by Microsoft as part of its .NET platform. It is used to develop various applications, including web, desktop, and mobile apps. The .NET platform is a framework that provides libraries, tools, and runtime environment for building applications with C# and other languages.
* Why learn C#? C# is a powerful and versatile language that is widely used in enterprise-level software development, games (using Unity), and more.
* Setting up the development environment: This section will guide users through installing Visual Studio or Visual Studio Code, along with the necessary .NET SDK.

## 2. C# basics: syntax, variables, and data types

* Understanding basic C# syntax: This section will introduce fundamental syntax rules, including namespaces, classes, methods (such as the Main method), and the use of the Console.WriteLine() method for outputting text.
* Variables and data types: This section will delve into the concept of variables for storing data, along with various built-in data types like integers (int), floating-point numbers (float, double), Booleans (bool), and characters (char). The use of the var keyword for implicitly typed local variables will also be covered.
* Naming conventions: Best practices for naming variables, methods, and classes will be discussed, emphasizing clarity and readability.

## 3. Control flow: making decisions and looping

* Conditional statements: This section will explore the use of if, else if, and else statements for executing different code blocks based on conditions.
* Switch statements: An introduction to switch statements for handling multiple conditions in a concise manner.
* Looping constructs: This section will cover different types of loops, such as for, while, do-while, and foreach, for repeating code blocks.
* Break and continue statements: These statements, used to control the flow of loops, will be introduced with examples.

## 4. Object-oriented programming (OOP) in C#

* Classes and objects: The foundational concepts of OOP, including classes as blueprints and objects as instances of classes, will be explained with relevant examples.
* Encapsulation: The concept of bundling data and methods within a class and controlling access through access modifiers will be explored.
* Inheritance: This section will cover the mechanism for creating new classes based on existing classes, promoting code reuse.
* Polymorphism: The ability of objects to take on different forms and respond to method calls in various ways will be explained.
* Abstraction: Defining essential characteristics while hiding implementation details through abstract classes and interfaces will be discussed.

## 5. Methods (functions)

* Defining and calling methods: This section will provide a detailed explanation of creating and invoking methods to encapsulate reusable blocks of code.
* Parameters and return values: Passing data to methods and receiving data back through return values will be covered.
* Method overloading: Defining multiple methods with the same name but different parameters will be introduced as a form of polymorphism.

## 6. Exception handling

* Understanding exceptions: This section will introduce the concept of exceptions as unexpected errors that occur during program execution.
* Using try-catch-finally blocks: The core constructs for handling exceptions, including try to identify code that might throw an exception, catch to handle the exception, and finally to ensure code execution regardless of whether an exception is thrown, will be explained with examples.
* Throwing exceptions: The use of the throw keyword to generate exceptions will be demonstrated.
* Best practices for exception handling: Guidance on how and when to use exceptions effectively will be provided, drawing from sources like.

## 7. File I/O (input/output)

* Reading from and writing to files: This section will introduce the basic operations of reading and writing data to files using classes from the System.IO namespace.
* Using StreamReader and StreamWriter: These classes for efficient reading and writing of text data to and from files will be covered.
* Binary file operations (optional): Briefly introduce how to handle binary files using classes like File.ReadAllBytes() and BinaryReader/BinaryWriter.

## 8. Putting it all together: Building a simple console application

* A step-by-step example: This section will guide the reader through building a simple C# console application, such as a calculator or a to-do list manager, integrating the concepts learned in previous sections.
* Input and output from the console: The Console.ReadLine() and Console.WriteLine() methods for user interaction will be used.

## 9. Next steps and resources

* Learning beyond the basics: Suggest exploring more advanced topics like data structures, collections, LINQ, asynchronous programming, web development with ASP.NET Core, and game development with Unity.
* Recommended learning resources: Point to official documentation (like [Microsoft Learn](https://www.google.com/url?sa=i&source=web&rct=j&url=https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/&ved=2ahUKEwjL9aT_rL-OAxWDs4QIHVK5EScQy_kOegYIAwgAEDI&opi=89978449&cd&psig=AOvVaw3AGxTN2Xbkg3veJCEaG3gm&ust=1752685653116000)), online tutorials, books, coding practice platforms, and communities for continued learning.

Contents

[Learning C# 2](#_Toc203548041)

[1. Introduction to C# 2](#_Toc203548042)

[2. C# basics: syntax, variables, and data types 2](#_Toc203548043)

[3. Control flow: making decisions and looping 2](#_Toc203548044)

[4. Object-oriented programming (OOP) in C# 3](#_Toc203548045)

[5. Methods (functions) 3](#_Toc203548046)

[6. Exception handling 3](#_Toc203548047)

[7. File I/O (input/output) 4](#_Toc203548048)

[8. Putting it all together: Building a simple console application 4](#_Toc203548049)

[9. Next steps and resources 4](#_Toc203548050)

[1. Introduction to C# 7](#_Toc203548051)

[What is C# and its place in the .NET ecosystem? 7](#_Toc203548052)

[Why learn C#? 7](#_Toc203548053)

[Getting Started with C# in Visual Studio Code 8](#_Toc203548054)

[Installing the C# Dev Kit 8](#_Toc203548055)

[Compiling & Running C# Programs in VS Code 8](#_Toc203548056)

[2. C# basics: syntax, variables, and data types 10](#_Toc203548057)

[Understanding basic C# syntax 10](#_Toc203548058)

[3. Control flow: making decisions and looping 15](#_Toc203548059)

[If statements 15](#_Toc203548060)

[If-else statements 15](#_Toc203548061)

[Else-if statements 16](#_Toc203548062)

[Switch statements 17](#_Toc203548063)

[For loops 18](#_Toc203548064)

[While loops 19](#_Toc203548065)

[Do-while loops 20](#_Toc203548066)

[Break and continue statements 21](#_Toc203548067)

[Continue statement 22](#_Toc203548068)

[4. Object-Oriented Programming (OOP) in C# 23](#_Toc203548069)

[Classes and objects 23](#_Toc203548070)

[Encapsulation 24](#_Toc203548071)

[Inheritance 25](#_Toc203548072)

[Polymorphism 26](#_Toc203548073)

[Abstraction 26](#_Toc203548074)

[5. Methods (functions) 28](#_Toc203548075)

[Defining and calling methods 28](#_Toc203548076)

[Method overloading 31](#_Toc203548077)

[6. Exception handling 32](#_Toc203548078)

[Understanding exceptions 32](#_Toc203548079)

[Throwing exceptions 34](#_Toc203548080)

[Custom exceptions 34](#_Toc203548081)

[7. File I/O (input/output) 37](#_Toc203548082)

[Reading from and writing to files 37](#_Toc203548083)

[Reading text files 37](#_Toc203548084)

[Writing text files 39](#_Toc203548085)

[Appending to text files 40](#_Toc203548086)

[Binary file operations (optional) 40](#_Toc203548087)

[8. Putting it all together: Building a simple console application 43](#_Toc203548088)

[1. Project setup 43](#_Toc203548089)

[2. Design the Task class 43](#_Toc203548090)

[3. Implement the main application logic (Program.cs) 44](#_Toc203548091)

[4. Code explanation 47](#_Toc203548092)

[5. Running the application 48](#_Toc203548093)

[9. Next steps and resources 49](#_Toc203548094)

[Recommended learning resources 51](#_Toc203548095)

[Continuing your journey 51](#_Toc203548096)

# 1. Introduction to C#

## What is C# and its place in the .NET ecosystem?

C# (pronounced "C sharp") is a modern, object-oriented programming language developed by Microsoft. It's a key part of the .NET ecosystem, which is a powerful and comprehensive software development platform. Think of C# as the language you use to write instructions, and .NET as the set of tools and rules that allow those instructions to be understood and executed across various devices and platforms.

Here's a breakdown:

* .NET is a software framework with a runtime environment (Common Language Runtime - CLR) and libraries for building applications.
* The CLR manages code execution, memory, and security.
* The .NET platform supports various application types, including desktop, web, mobile, cloud, games, and IoT applications.

## Why learn C#?

Key reasons to learn C# include its versatility as a general-purpose language, its Object-Oriented (OOP) nature, and its relative ease of learning. It benefits from strong community support, high demand in the industry, and features that enhance developer productivity. Furthermore, C# and .NET are now open source and cross-platform.

Setting up the development environment

To begin writing C# code, you'll need to set up your development environment. The most common tools are Microsoft Visual Studio or Visual Studio Code.

1. Installing Visual Studio (Windows)

* Download and run the installer from the official Microsoft website.
* Select the .NET desktop development workload.
* Follow the installation prompts.
* Launch Visual Studio and configure preferences if prompted.

2. Installing Visual Studio Code (Cross-platform)

* Download and install Visual Studio Code.
* In VS Code, open Extensions and search for "C# Dev Kit".
* Install the extension and follow instructions to install the latest .NET SDK.
* Restart VS Code as needed.

## Getting Started with C# in Visual Studio Code

To develop C# applications smoothly in Visual Studio Code (VS Code), you’ll want to install the C# Dev Kit extension, which enhances IntelliSense, debugging, project management, and code navigation.

## Installing the C# Dev Kit

Open Extensions Panel Launch VS Code and click on the Extensions icon in the Activity Bar (or press Ctrl+Shift+X).

Search for C# Dev Kit Type C# Dev Kit into the search bar.

Install the Extension Select the C# Dev Kit extension by Microsoft and click Install. This will also install required dependencies like the C# for Visual Studio Code extension and the C# Debugger.

Reload or Restart VS Code After installation, restart VS Code to ensure everything initializes properly.

Once installed, you'll get an enriched coding experience with Solution Explorer, dynamic debug configurations, and run/debug buttons directly in your editor.

## Compiling & Running C# Programs in VS Code

Create a Console Project Open the terminal in VS Code and run:

bash

dotnet new console -o HelloWorld

cd HelloWorld

code .

Run the Program

Open Program.cs

Press F5, or click the play/debug icon in the top-right corner

Your program launches and shows output in the terminal or Debug Console

Sample Code

csharp

using System;

class Program {

static void Main() {

Console.WriteLine("Hello from VS Code with C# Dev Kit!");

}

}

🧵 Compiling & Running from the Command Prompt

Create the Project Manually

bash

dotnet new console -o CmdApp

cd CmdApp

Run the App

bash

dotnet run

Output Your console will display:

Hello from VS Code with C# Dev Kit!

Once set up, you're ready to write your first C# programs. The next chapter will cover basic C# syntax.

# 2. C# basics: syntax, variables, and data types

Now that your C# development environment is set up, let's dive into the core building blocks of any C# program. This chapter will introduce you to basic syntax rules, how to store and manipulate data using variables and data types, and important naming conventions for writing clear and maintainable code.

## Understanding basic C# syntax

C# follows a syntax that is similar to other C-style languages like C++, Java, and JavaScript. If you have experience with these languages, many aspects of C# syntax will feel familiar.

* Case Sensitivity: C# is case-sensitive, meaning that myVariable and MyVariable are treated as two different entities by the compiler.
* Statements and Semicolons: Every statement in C# must end with a semicolon (;). This tells the compiler that the instruction is complete. For example:

csharp

int age = 30; *// Declaration and assignment*

Console.WriteLine(age); *// Method call*

* Code Blocks and Curly Braces: Curly braces ({}) define blocks of code, used for grouping statements together in classes, methods, conditional statements, and loops.
* Whitespace and Indentation: C# generally ignores extra spaces and newlines (whitespace). However, using proper indentation and consistent whitespace makes your code significantly more readable and easier to understand.
* Namespaces: Namespaces are used to organize code and prevent naming conflicts, especially in larger projects. You'll often see the using System; statement at the top of C# files, allowing you to use classes like Console without fully qualifying them as System.Console.
* Classes: A C# program is built around classes, which are blueprints for creating objects. You'll encounter classes like Console, which provides methods for interacting with the console window.
* Methods: Methods are blocks of code that perform specific tasks. The Main method is the entry point of a C# console application, where program execution begins.
* Console.WriteLine(): This is a fundamental method used to display output on the console. It automatically adds a new line after the output. For example:

csharp

Console.WriteLine("Hello, C#!"); // Displays "Hello, C#!" and moves to the next line

Console.WriteLine(123); // Displays the number 123

There is also a Console.Write() method that prints output without adding a new line, allowing for continuous output on the same line.

Variables and data types

Variables are essentially containers for storing data values. In C#, when you declare a variable, you must specify its data type, which determines the kind of values it can hold and the operations that can be performed on it. C# is a strongly-typed language, meaning you must explicitly declare the type of a variable.

Declaring and initializing variables

To declare a variable, you specify its data type followed by the variable name. You can optionally assign an initial value when you declare it (initialization).

csharp

// Declaring a variable

int myNumber;

// Assigning a value to the variable

myNumber = 10;

// Declaring and initializing a variable in one step

string greeting = "Hello, World!";

Common built-in data types

C# provides a variety of built-in data types for different kinds of data:

* int (Integer): Used for storing whole numbers (positive or negative) without decimals. It's a 32-bit integer.

csharp

int score = 100;

int temperature = -5;

* float (Single-precision Floating-Point): Used for storing fractional numbers with decimals, offering a precision of 6 to 7 decimal digits. It's a 32-bit floating-point number and requires an f suffix for literal values.

csharp

float pi = 3.14f;

* double (Double-precision Floating-Point): Also used for storing fractional numbers, but with a higher precision of 15 to 16 decimal digits. It's a 64-bit floating-point number and is the default for real numbers.

csharp

double price = 19.99;

* bool (Boolean): Stores a logical value of either true or false.

csharp

bool isActive = true;

bool hasPermission = false;

* char (Character): Stores a single character (letter, number, or symbol), enclosed in single quotes (').

csharp

char initial = 'J';

* string (String): Stores a sequence of characters (text), enclosed in double quotes (").

csharp

string name = "Alice";

string message = "Hello, C# World!";

csharp

// Example using different data types

int age = 25;

double height = 1.75;

char gender = 'M';

string city = "New York";

bool isStudent = true;

Console.WriteLine($"Name: {name}, Age: {age}, Height: {height}, Gender: {gender}, City: {city}, Student: {isStudent}");

// Output: Name: Alice, Age: 25, Height: 1.75, Gender: M, City: New York, Student: True

The var keyword (implicitly typed local variables)

C# 3.0 introduced the var keyword for declaring implicitly typed local variables. When you use var, the compiler infers the variable's type from the initialization expression on the right-hand side.

csharp

var myImplicitNumber = 10; *// Compiler infers 'int'*

var myImplicitString = "Hello"; *// Compiler infers 'string'*

It's important to understand that var does not make the variable dynamically typed. The type is still determined at compile-time, it's just not explicitly written by the programmer. Using var can improve readability in some situations, especially with complex type names. However, it should be used judiciously to avoid making the code less clear.

Naming conventions

Consistent naming conventions are essential for writing readable and maintainable C# code. Here are some commonly accepted conventions:

* Classes and Methods: Use PascalCase (each word capitalized).
  + MyClass
  + CalculateTotal()
* Local Variables and Method Parameters: Use camelCase (first word lowercase, subsequent words capitalized).
  + myVariable
  + firstName
* Private Fields (Class-level variables): Often use camelCase, sometimes with an underscore prefix (\_).
  + \_customerName
  + \_totalOrders
* Constants: Use PascalCase for const and static readonly fields.
  + MaxOrderLimit
* Interfaces: Prefix with the letter I and use PascalCase.
  + IDisposable
  + IComparable

Adhering to these conventions makes code easier to understand and work with. The next chapter will explore control flow, allowing programs to make decisions and repeat actions.

# 3. Control flow: making decisions and looping

In programming, control flow determines the order in which statements are executed. It allows your programs to make decisions, repeat actions, and respond dynamically to different situations. This chapter covers the fundamental control flow constructs in C#: conditional statements for making decisions and looping constructs for repetitive tasks.

Conditional statements: making decisions

Conditional statements allow your program to execute different blocks of code based on whether a specific condition evaluates to true or false.

## If statements

The if statement is the most basic conditional statement. It executes a block of code only if its condition is true.

* Syntax:

csharp

if (condition)

{

*// Code to be executed if the condition is true*

}

Where condition is a Boolean expression that evaluates to true or false.

* Example:

csharp

int score = 85;

if (score >= 60)

{

Console.WriteLine("You passed the exam!");

}

## If-else statements

The if-else statement allows you to choose between two alternative code paths based on a Boolean expression.

* Syntax:

csharp

if (condition)

{

*// Code for when the condition is true*

}

else

{

*// Code for when the condition is false*

}

* Example:

csharp

int age = 17;

if (age >= 18)

{

Console.WriteLine("You are eligible to vote.");

}

else

{

Console.WriteLine("You are not yet eligible to vote.");

}

## Else-if statements

Else if clauses are used to test multiple conditions in sequence. These are evaluated only if the preceding if or else if conditions are false.

* Syntax:

csharp

if (condition1)

{

*// Code for condition1 is true*

}

else if (condition2)

{

*// Code for condition2 is true (and condition1 is false)*

}

else

{

*// Code for all previous conditions are false*

}

* Example:

csharp

int time = 22;

if (time < 10)

{

Console.WriteLine("Good morning.");

}

else if (time < 20)

{

Console.WriteLine("Good day.");

}

else

{

Console.WriteLine("Good evening.");

} *// Outputs "Good evening."*

## Switch statements

The switch statement provides a concise way to handle multiple possible execution paths based on the value of a variable or expression. It can replace long if-else if chains.

* Syntax:

csharp

switch (expression)

{

case value1:

*// Code to execute if expression equals value1*

break;

case value2:

*// Code to execute if expression equals value2*

break;

*// ... more cases*

default:

*// Code to execute if no case matches (optional)*

break;

}

Each case block requires a break; statement at the end to prevent "fall-through" into the next case, unless multiple cases are deliberately grouped to execute the same code.

* Example:

csharp

char grade = 'B';

switch (grade)

{

case 'A':

Console.WriteLine("Excellent!");

break;

case 'B':

Console.WriteLine("Good job!");

break;

case 'C':

Console.WriteLine("You passed.");

break;

default:

Console.WriteLine("Needs improvement.");

break;

}

In C#, [Learn Microsoft notes](https://www.google.com/url?sa=i&source=web&rct=j&url=https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/selection-statements&ved=2ahUKEwiw0O-Orr-OAxWpRzABHVVsDawQy_kOegYIAwgAEDE&opi=89978449&cd&psig=AOvVaw1mckXODDfde3kGpQ89UjA0&ust=1752685954209000) the switch statement can be used with various types, including integral types, enums, strings, and even type patterns.

Looping constructs: repeating actions

Loops allow you to repeatedly execute a block of code until a certain condition is met.

## For loops

The for loop is ideal when you know exactly how many times you want to iterate or when you need to use a counter.

* Syntax:

csharp

for (initialization; condition; increment/decrement)

{

*// Code to be executed in each iteration*

}

* + initialization: Executed once before the loop starts, typically used to declare and initialize a loop counter variable.
  + condition: Evaluated before each iteration. If true, the loop continues; if false, the loop terminates.
  + increment/decrement: Executed after each iteration, typically used to update the loop counter.
* Example:

csharp

for (int i = 0; i < 5; i++)

{

Console.WriteLine($"Current number: {i}");

}

*// Output:*

*// Current number: 0*

*// Current number: 1*

*// Current number: 2*

*// Current number: 3*

*// Current number: 4*

## While loops

The while loop repeatedly executes a block of code as long as its condition remains true.

* Syntax:

csharp

while (condition)

{

*// Code to be executed while the condition is true*

}

Ensure that the condition eventually becomes false to avoid an infinite loop.

* Example:

csharp

int count = 0;

while (count < 3)

{

Console.WriteLine($"Count is: {count}");

count++; *// Increment the counter*

}

*// Output:*

*// Count is: 0*

*// Count is: 1*

*// Count is: 2*

## Do-while loops

The do-while loop is similar to the while loop, but the code block is executed at least once before the condition is evaluated.

* Syntax:

csharp

do

{

*// Code to be executed at least once*

} while (condition);

* Example:

csharp

int num = 5;

do

{

Console.WriteLine($"Number: {num}");

num--;

} while (num > 5); *// The loop runs once, then the condition is false.*

*// Output:*

*// Number: 5*

Foreach loops

The foreach loop is designed for iterating over elements in collections (like arrays and lists) without explicitly managing an index. This is often simpler and more readable than a for loop for this purpose.

* Syntax:

csharp

foreach (dataType variableName in collectionName)

{

*// Code to be executed for each item*

}

* + dataType: The type of the elements in the collection.
  + variableName: A variable to hold the current element during each iteration.
  + collectionName: The collection to iterate over (e.g., an array, a List<T>).
* Example:

csharp

string[] names = { "Alice", "Bob", "Charlie" };

foreach (string name in names)

{

Console.WriteLine($"Hello, {name}!");

}

*// Output:*

*// Hello, Alice!*

*// Hello, Bob!*

*// Hello, Charlie!*

[Dev Leader mentions](https://www.google.com/url?sa=i&source=web&rct=j&url=https://www.devleader.ca/2024/01/03/understanding-foreach-loops-in-c-what-you-need-to-know&ved=2ahUKEwiw0O-Orr-OAxWpRzABHVVsDawQy_kOegYIAwgAEFU&opi=89978449&cd&psig=AOvVaw1mckXODDfde3kGpQ89UjA0&ust=1752685954209000) foreach loops manage the iteration logic automatically, leading to cleaner code and easier debugging compared to for loops.

## Break and continue statements

These statements provide additional control within loops.

Break statement

The break statement immediately terminates the innermost loop (or switch statement) it's contained within, transferring control to the statement following the loop.

* Example:

csharp

for (int i = 0; i < 10; i++)

{

if (i == 5)

{

Console.WriteLine("Found 5, breaking the loop.");

break; *// Exit the loop when i is 5*

}

Console.WriteLine($"Current i: {i}");

}

*// Output:*

*// Current i: 0*

*// Current i: 1*

*// Current i: 2*

*// Current i: 3*

*// Current i: 4*

*// Found 5, breaking the loop.*

## Continue statement

The continue statement skips the rest of the current iteration of the loop and proceeds to the next iteration (checking the loop condition again).

* Example:

csharp

for (int i = 0; i < 5; i++)

{

if (i == 2)

{

Console.WriteLine("Skipping 2.");

continue; *// Skip the rest of the code in this iteration if i is 2*

}

Console.WriteLine($"Processing number: {i}");

}

*// Output:*

*// Processing number: 0*

*// Processing number: 1*

*// Skipping 2.*

*// Processing number: 3*

*// Processing number: 4*

Mastering conditional and looping constructs is fundamental to writing effective C# programs. They allow code to respond to different inputs, automate repetitive tasks, and implement complex logic. The next chapter will introduce Object-Oriented Programming (OOP) in C#.

# 4. Object-Oriented Programming (OOP) in C#

Object-Oriented Programming (OOP) is a powerful programming paradigm that organizes software design around objects rather than functions and logic. C# is built from the ground up as an object-oriented language, making OOP principles central to writing effective and maintainable C# applications.

This chapter introduces the fundamental concepts of OOP in C#, often referred to as the four pillars: encapsulation, inheritance, polymorphism, and abstraction. Understanding these concepts is crucial for building robust, scalable, and modular applications.

## Classes and objects

* Classes: A class is a blueprint or template for creating objects. It defines the characteristics (data or properties) and behaviors (methods or functions) that objects of that class will possess. Think of a class like a blueprint for a house: it describes what the house will have (rooms, doors, windows) but it's not the actual house itself.

csharp

public class Dog

{

*// Fields (characteristics)*

public string Name;

public string Breed;

*// Method (behavior)*

public void Bark()

{

Console.WriteLine($"{Name} says Woof!");

}

}

* Objects: An object is an instance of a class. It's a concrete entity created from the class blueprint, with its own unique set of data values. Continuing the analogy, an object would be an actual house built according to the blueprint, with its own specific color, size, and location. You can create multiple objects from a single class, each with its own state.

csharp

// Creating objects (instances) of the Dog class

Dog myDog = new Dog(); // 'myDog' is an object

myDog.Name = "Buddy";

myDog.Breed = "Golden Retriever";

myDog.Bark(); // Output: Buddy says Woof!

Dog anotherDog = new Dog(); // 'anotherDog' is another object

anotherDog.Name = "Lucy";

anotherDog.Breed = "Labrador";

anotherDog.Bark(); // Output: Lucy says Woof!

## Encapsulation

Encapsulation is the principle of bundling data and methods that operate on the data within a single unit, which is the class. It involves hiding the internal state of an object and only exposing necessary information through a controlled interface, promoting data integrity. This is typically achieved using access modifiers like public, private, and protected. For example, in a BankAccount class, the balance field can be private to prevent direct modification, with public methods like Deposit() and Withdraw() used for interaction.

csharp

public class BankAccount

{

private decimal \_balance; *// Private field*

public BankAccount(decimal initialBalance)

{

\_balance = initialBalance;

}

public void Deposit(decimal amount)

{

if (amount > 0)

{

\_balance += amount;

Console.WriteLine($"Deposited: {amount}. New balance: {\_balance}");

}

}

public void Withdraw(decimal amount)

{

if (amount > 0 && \_balance >= amount)

{

\_balance -= amount;

Console.WriteLine($"Withdrew: {amount}. New balance: {\_balance}");

}

else

{

Console.WriteLine("Insufficient funds or invalid amount.");

}

}

public decimal Balance

{

get { return \_balance; }

}

}

## Inheritance

Inheritance allows creating new classes (derived classes) based on existing classes (base classes). This provides code reusability as derived classes inherit members from the base class and can also add or override them. Inheritance is indicated using the colon (:) symbol. For instance, a Car and Bicycle can inherit from a Vehicle base class.

csharp

public class Vehicle *// Base class*

{

public string Brand { get; set; }

public int Year { get; set; }

public virtual void Start()

{

Console.WriteLine("Vehicle is starting.");

}

}

public class Car : Vehicle *// Derived class*

{

public int NumberOfDoors { get; set; }

public override void Start()

{

Console.WriteLine("Car is starting with ignition.");

}

public void Accelerate()

{

Console.WriteLine("Car is accelerating.");

}

}

[Learn Microsoft notes](https://www.google.com/url?sa=i&source=web&rct=j&url=https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/inheritance&ved=2ahUKEwid7qm-rr-OAxWbSTABHYDRJdsQy_kOegYIAwgAECM&opi=89978449&cd&psig=AOvVaw0WqhxChwIHmHCQgDyCUq_0&ust=1752686053743000) C# does not support multiple inheritance of classes.

## Polymorphism

Polymorphism means "many forms". It allows objects of different classes to be treated as a common base type, responding to the same method call in different ways. It provides a single interface for multiple underlying forms.

1. Method Overriding (Runtime Polymorphism): Achieved with virtual and override keywords. The method implementation is determined at runtime based on the object type. [Medium mentions](https://www.google.com/url?sa=i&source=web&rct=j&url=https://mvineetsharma.medium.com/dynamic-polymorphism-in-net-core-with-examples-e1f7274672c4&ved=2ahUKEwid7qm-rr-OAxWbSTABHYDRJdsQy_kOegYIAwgAECY&opi=89978449&cd&psig=AOvVaw0WqhxChwIHmHCQgDyCUq_0&ust=1752686053743000)

csharp

Vehicle vehicle1 = new Car { Brand = "Honda" };

vehicle1.Start(); *// Calls Car's Start method at runtime*

1. Method Overloading (Compile-time Polymorphism): Defining methods with the same name but different parameter lists within the same class. [Stackify says](https://www.google.com/url?sa=i&source=web&rct=j&url=https://stackify.com/oop-concepts-c-sharp/&ved=2ahUKEwid7qm-rr-OAxWbSTABHYDRJdsQy_kOegYIAwgAECo&opi=89978449&cd&psig=AOvVaw0WqhxChwIHmHCQgDyCUq_0&ust=1752686053743000" \t "_blank) The compiler chooses the correct method based on arguments at compile-time.

csharp

public class Calculator

{

public int Add(int a, int b) { return a + b; }

public double Add(double a, double b) { return a + b; } *// Overloaded*

}

## Abstraction

Abstraction hides complex details and shows only essential features. It simplifies interaction by providing a generalized view . Abstraction is achieved using:

1. Abstract Classes: Cannot be instantiated directly but can contain abstract methods (no implementation, must be implemented by derived classes) and non-abstract methods.

csharp

public abstract class Shape *// Abstract class*

{

public abstract double GetArea(); *// Abstract method*

}

public class Circle : Shape

{

public override double GetArea() { */\* Implementation \*/* return 0;}

}

1. Interfaces: Define a contract with method signatures, properties, etc., that implementing classes must adhere to. A class can implement multiple interfaces.

csharp

public interface IDisplayable *// Interface*

{

void DisplayInfo(); *// Method declaration*

}

public class Product : IDisplayable *// Implements interface*

{

public void DisplayInfo() { */\* Implementation \*/* }

}

[InfoWorld says](https://www.google.com/url?sa=i&source=web&rct=j&url=https://www.infoworld.com/article/2242358/when-to-use-an-abstract-class-vs-interface-in-csharp.html&ved=2ahUKEwid7qm-rr-OAxWbSTABHYDRJdsQy_kOegYIAwgAEDQ&opi=89978449&cd&psig=AOvVaw0WqhxChwIHmHCQgDyCUq_0&ust=1752686053743000) The choice between abstract classes and interfaces depends on design needs.

These four principles are key to building well-structured C# applications.

# 5. Methods (functions)

In C#, methods are blocks of code that perform specific tasks. They are a fundamental building block of organized, reusable, and modular programming. Methods allow you to encapsulate a sequence of operations into a named unit, which can then be called from different parts of your program.

## Defining and calling methods

A method is defined within a class, struct, or interface and consists of a signature and a body.

Method signature

The method signature specifies:

* Access level: Determines where the method can be accessed (e.g., public, private, protected). Default is private.
* Optional modifiers: Keywords like abstract or sealed that modify the method's behavior.
* Return value type: The data type of the value the method will return, or void if it doesn't return anything.
* Method name: A unique name that identifies the method.
* Parameters: A list of variables, defined within parentheses, that receive input values from the caller. Empty parentheses indicate no parameters.

csharp

// Method that returns a string and takes two string parameters

public string GetFullName(string firstName, string lastName)

{

return $"{firstName} {lastName}";

}

// Method that returns nothing (void) and takes no parameters

public void SayHello()

{

Console.WriteLine("Hello!");

}

// Method that returns an int and takes one int parameter

private int Square(int number)

{

return number \* number;

}

Calling (executing) methods

To call a method, use its name followed by parentheses (). If the method belongs to an object, you call it using the object name followed by a dot (.). If it's a static method, call it using the class name.

csharp

// Calling a method that returns a value and storing it

string fullName = GetFullName("John", "Doe");

Console.WriteLine(fullName); // Output: John Doe

// Calling a method that doesn't return a value

SayHello(); // Output: Hello!

// Calling a static method of the Console class

Console.WriteLine("This is a static method call.");

// Example of calling a method on an object instance (if Square was public)

// MyClass myObject = new MyClass(); // Assuming MyClass contains the Square method

// int squaredNumber = myObject.Square(5);

// Console.WriteLine(squaredNumber); // Output: 25

Parameters and return values

Methods can accept input values through parameters and send back results using return values.

* Parameters: Variables declared in the method's signature that receive values (arguments) when the method is called. Parameters make methods flexible and reusable.
  + Value Parameters: The most common type; a copy of the argument's value is passed. Changes within the method do not affect the original argument.
  + Reference Parameters (ref, out): Passed by reference, meaning the method operates directly on the original memory location. Changes inside the method do affect the original variable.
    - ref: The argument must be initialized before being passed. The method can modify it.
    - out: The argument does not need to be initialized before being passed, but the method must assign a value to it before exiting. Often used to return multiple values.
* Return Values: If the method's return type is not void, it can return a value using the return statement. The value returned must match the specified return type.

csharp

// Example with parameters, ref, out, and return values

public class Calculator

{

// Method that uses value parameters and returns an int

public int Add(int a, int b)

{

return a + b;

}

// Method that uses a ref parameter

public void MultiplyByTwo(ref int number)

{

number \*= 2;

}

// Method that uses an out parameter to return two values

public void GetMinMax(int[] numbers, out int min, out int max)

{

min = numbers.Min();

max = numbers.Max();

}

}

// In your Main method or another class:

Calculator calc = new Calculator();

int sum = calc.Add(5, 3); // sum will be 8

Console.WriteLine($"Sum: {sum}");

int val = 10;

calc.MultiplyByTwo(ref val); // 'val' is passed by reference

Console.WriteLine($"Value after multiplication: {val}"); // Output: Value after multiplication: 20

int[] myNumbers = { 10, 4, 25, 7 };

int minimum; // No initialization needed for 'out' parameter

int maximum;

calc.GetMinMax(myNumbers, out minimum, out maximum);

Console.WriteLine($"Min: {minimum}, Max: {maximum}"); // Output: Min: 4, Max: 25

## Method overloading

Method overloading is a form of polymorphism in C#. It allows a class to have multiple methods with the same name, as long as their parameter lists differ. The differences can be in the:

* Number of parameters
* Data types of parameters
* Order of parameters

csharp

public class Printer

{

*// Overload 1: Prints an integer*

public void Print(int number)

{

Console.WriteLine($"Integer: {number}");

}

*// Overload 2: Prints a string*

public void Print(string text)

{

Console.WriteLine($"String: {text}");

}

*// Overload 3: Prints a double*

public void Print(double value)

{

Console.WriteLine($"Double: {value}");

}

*// Overload 4: Prints an integer and a message*

public void Print(int number, string message)

{

Console.WriteLine($"{message}: {number}");

}

}

*// Usage:*

Printer printer = new Printer();

printer.Print(123); *// Calls Overload 1*

printer.Print("Hello"); *// Calls Overload 2*

printer.Print(3.14); *// Calls Overload 3*

printer.Print(456, "My number"); *// Calls Overload 4*

Note that the return type alone is not sufficient to overload a method. Overloaded methods may have different return types, but their parameter lists must differ. [Code Maze also mentions](https://www.google.com/url?sa=i&source=web&rct=j&url=https://code-maze.com/csharp-method-overloading/&ved=2ahUKEwjt9q_Zrr-OAxUomYQIHXw1GwUQy_kOegYIAwgAEDo&opi=89978449&cd&psig=AOvVaw11jrl83QIS4baetQ4aJ42_&ust=1752686110469000)

Mastering methods is a key step towards writing modular, maintainable, and efficient C# code. The next chapter will explore exception handling, a crucial aspect of building robust applications.

# 6. Exception handling

Even the most carefully written code can encounter unexpected issues during execution. These unexpected events, or exceptions, can disrupt the normal flow of your program and potentially lead to crashes. Exception handling in C# is a mechanism that allows you to anticipate, detect, and respond to these exceptions in a controlled manner, preventing abrupt termination and ensuring your application remains robust.

## Understanding exceptions

An exception is an object that encapsulates information about an error or an abnormal event that occurred during the execution of a program. When such an event occurs, an exception is thrown. If this thrown exception is not handled, it will ultimately terminate the program.

C# exceptions are represented by classes, and the base class for all exceptions is System.Exception. There are numerous predefined exception types for common scenarios, such as DivideByZeroException when attempting to divide by zero, or NullReferenceException when trying to access a member on an object that is null.

Using try-catch-finally blocks

The core of exception handling in C# revolves around the try-catch-finally block.

* try block: Encloses the code that might potentially throw an exception.
* catch block: Contains code to handle a specific type of exception thrown within the try block.
* finally block: Contains code that is guaranteed to execute, regardless of whether an exception occurred or was handled.

csharp

try

{

*// Code that might cause an exception*

int a = 10;

int b = int.Parse(Console.ReadLine()); *// Potentially throws FormatException*

int result = a / b; *// Potentially throws DivideByZeroException*

Console.WriteLine($"Result: {result}");

}

catch (DivideByZeroException ex)

{

Console.WriteLine($"Error: Cannot divide by zero. {ex.Message}"); *// Catches specific exception*

}

catch (FormatException ex)

{

Console.WriteLine($"Error: Invalid number format. {ex.Message}"); *// Catches specific exception*

}

catch (Exception ex) *// Catches any other type of exception (more general)*

{

Console.WriteLine($"An unexpected error occurred: {ex.Message}");

}

finally

{

Console.WriteLine("This block always executes, regardless of exceptions."); *// Clean up resources, etc.*

}

* Execution Flow:
  + If no exception occurs in the try block, the catch blocks are skipped, and the finally block executes.
  + If an exception occurs, the try block's execution stops at the point where the exception is thrown. The CLR then searches for the first catch block that can handle that type of exception.
  + The matching catch block executes, and then the finally block executes.
  + If no matching catch block is found, the exception will be unhandled, potentially causing the program to terminate.

## Throwing exceptions

You can explicitly throw an exception using the throw keyword. This is useful when you detect a condition that prevents your method from completing its task or when an object is in an invalid state.

csharp

public void ValidateAge(int age)

{

if (age < 0 || age > 120)

{

*// Throw an ArgumentOutOfRangeException if the age is invalid*

throw new ArgumentOutOfRangeException(nameof(age), "Age must be between 0 and 120.");

}

Console.WriteLine($"Age {age} is valid.");

}

*// Usage:*

try

{

ValidateAge(-5);

}

catch (ArgumentOutOfRangeException ex)

{

Console.WriteLine($"Validation Error: {ex.Message}");

}

## Custom exceptions

While predefined exceptions cover many scenarios, you can also create your own custom exception types by inheriting from System.Exception or one of its derived classes. This allows you to create more specific, meaningful error types tailored to your application's domain.

csharp

public class InsufficientFundsException : Exception *// Inherit from System.Exception*

{

public decimal RequestedAmount { get; }

public decimal CurrentBalance { get; }

public InsufficientFundsException(decimal requestedAmount, decimal currentBalance)

: base($"Attempted to withdraw {requestedAmount:C} but only {currentBalance:C} is available.")

{

RequestedAmount = requestedAmount;

CurrentBalance = currentBalance;

}

}

*// Usage in a BankAccount class method:*

public void Withdraw(decimal amount)

{

if (amount > 0 && \_balance >= amount)

{

\_balance -= amount;

Console.WriteLine($"Withdrew: {amount}. New balance: {\_balance}");

}

else if (amount > \_balance)

{

throw new InsufficientFundsException(amount, \_balance);

}

else

{

throw new ArgumentException("Withdrawal amount must be positive.");

}

}

Best practices for exception handling

* Catch specific exceptions first: Handle the most specific exception types (e.g., DivideByZeroException) before more general ones (e.g., Exception).
* Avoid catching System.Exception generally: Only use it at the highest level of your application to catch any unexpected errors before the program terminates. Avoid catching it lower in the call stack, [Learn Microsoft advises](https://www.google.com/url?sa=i&source=web&rct=j&url=https://learn.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions&ved=2ahUKEwi-kbbqrr-OAxUOVTABHZnWIjUQy_kOegYIAwgAECk&opi=89978449&cd&psig=AOvVaw3JBMEduEj1-jRBNYdRb6n9&ust=1752686146226000) unless you intend to rethrow the original exception or wrap it in a different, more appropriate exception type.
* Use finally for cleanup: Ensure resources (like file streams or database connections) are properly closed or disposed of, even if an exception occurs.
* Avoid empty catch blocks: Do not silently ignore exceptions. If you catch an exception, you should at least log it or provide meaningful feedback.
* Throw exceptions for truly exceptional circumstances: Avoid using exceptions for normal program flow control where an if statement or a return value would suffice.
* Log exceptions: Implement a logging mechanism to record exception details (message, stack trace, context) for debugging and analysis.
* Rethrow exceptions appropriately: If you catch an exception but cannot fully handle it at that level, you might need to rethrow it to allow a higher level in the call stack to address it. Use throw; to preserve the original stack trace. Avoid throw ex; as it resets the stack trace.
* Create custom exceptions for specific error scenarios: This makes your code more readable, maintainable, and provides clearer information about application-specific errors.

By effectively implementing exception handling, you can create more resilient C# applications that can gracefully recover from errors and provide a better user experience. The next chapter will explore file I/O operations.

# 7. File I/O (input/output)

Interacting with the file system is a crucial part of many applications, allowing them to persistently store and retrieve data. In C#, File I/O (Input/Output) operations are handled primarily through the System.IO namespace. This chapter introduces how to read from and write to text and binary files.

## Reading from and writing to files

C# offers various classes for handling file operations, each suited for different scenarios. You will use classes from the System.IO namespace. Remember to include using System.IO; at the top of your code file.

## Reading text files

1. File.ReadAllText(filePath): Reads the entire content of a text file into a single string. This is convenient for small files but can be inefficient for large files as it loads the entire file into memory at once.

csharp

try

{

string content = File.ReadAllText("myTextFile.txt");

Console.WriteLine("File content:\n" + content);

}

catch (FileNotFoundException)

{

Console.WriteLine("File not found.");

}

catch (Exception ex)

{

Console.WriteLine($"An error occurred: {ex.Message}");

}

1. File.ReadAllLines(filePath): Reads all lines from a text file into a string array, with each array element representing one line.

csharp

try

{

string[] lines = File.ReadAllLines("myTextFile.txt");

Console.WriteLine("File content (line by line):");

foreach (string line in lines)

{

Console.WriteLine(line);

}

}

catch (FileNotFoundException)

{

Console.WriteLine("File not found.");

}

catch (Exception ex)

{

Console.WriteLine($"An error occurred: {ex.Message}");

}

1. StreamReader: Used for efficiently reading large text files by providing a stream-based approach. It reads data incrementally, rather than loading the entire file into memory. It can read line by line using the ReadLine() method until the end of the file is reached (indicated by a null return). It's crucial to use StreamReader objects within a using statement to ensure that the stream is properly closed and disposed of, even if an error occurs.

csharp

string filePath = "largeTextFile.txt";

try

{

using (StreamReader sr = new StreamReader(filePath)) *// Using statement ensures disposal*

{

string line;

while ((line = sr.ReadLine()) != null)

{

Console.WriteLine(line);

}

}

}

catch (FileNotFoundException)

{

Console.WriteLine("File not found.");

}

catch (Exception ex)

{

Console.WriteLine($"An error occurred: {ex.Message}");

}

## Writing text files

1. File.WriteAllText(filePath, content): Writes the specified string content to a file. If the file exists, it will be overwritten. If the file does not exist, it will be created.

csharp

try

{

string contentToWrite = "This is a new line of text.";

File.WriteAllText("output.txt", contentToWrite);

Console.WriteLine("Content written to output.txt");

}

catch (Exception ex)

{

Console.WriteLine($"An error occurred while writing: {ex.Message}");

}

1. File.WriteAllLines(filePath, stringArray): Writes all lines from a string array to a file, with each array element becoming a new line. Like WriteAllText, this method will overwrite an existing file or create a new one.

csharp

try

{

string[] linesToWrite = { "First line.", "Second line.", "Third line." };

File.WriteAllLines("multiLineOutput.txt", linesToWrite);

Console.WriteLine("Lines written to multiLineOutput.txt");

}

catch (Exception ex)

{

Console.WriteLine($"An error occurred while writing lines: {ex.Message}");

}

1. StreamWriter: Used for efficiently writing to large text files or when you need more control over the writing process. Like StreamReader, it's best used with a using statement for proper resource management.

csharp

string filePath = "largeOutput.txt";

try

{

using (StreamWriter sw = new StreamWriter(filePath, append: true)) *// 'append: true' adds to the file, 'false' overwrites*

{

sw.WriteLine("This is the first line.");

sw.WriteLine("This is another line added to the file.");

}

Console.WriteLine("Content written using StreamWriter.");

}

catch (Exception ex)

{

Console.WriteLine($"An error occurred while using StreamWriter: {ex.Message}");

}

## Appending to text files

You can append text to existing files without overwriting the content using File.AppendAllText() or File.AppendAllLines().

csharp

string additionalContent = "This is additional content appended.";

File.AppendAllText("output.txt", additionalContent); *// Appends to the file*

Console.WriteLine("Content appended to output.txt");

## Binary file operations (optional)

For reading and writing binary data (like images, audio, or serialized objects), C# provides classes such as BinaryReader and BinaryWriter. These classes operate on primitive data types and read/write them in binary format. File.ReadAllBytes() and File.WriteAllBytes() are also useful for simple operations with byte arrays.

csharp

// Writing binary data

const string binaryFileName = "appSettings.dat";

try

{

using (var stream = File.Open(binaryFileName, FileMode.Create))

using (var writer = new BinaryWriter(stream, Encoding.UTF8, false))

{

writer.Write(123); // Write an integer

writer.Write(3.14159f); // Write a float

writer.Write("Hello"); // Write a string

writer.Write(true); // Write a boolean

}

Console.WriteLine("Binary data written to appSettings.dat.");

}

catch (Exception ex)

{

Console.WriteLine($"Error writing binary file: {ex.Message}");

}

// Reading binary data

try

{

if (File.Exists(binaryFileName))

{

using (var stream = File.Open(binaryFileName, FileMode.Open))

using (var reader = new BinaryReader(stream, Encoding.UTF8, false))

{

int intValue = reader.ReadInt32();

float floatValue = reader.ReadSingle();

string stringValue = reader.ReadString();

bool boolValue = reader.ReadBoolean();

Console.WriteLine($"Read from binary file: Int={intValue}, Float={floatValue}, String={stringValue}, Bool={boolValue}");

}

}

else

{

Console.WriteLine("Binary file not found.");

}

}

catch (Exception ex)

{

Console.WriteLine($"Error reading binary file: {ex.Message}");

}

Binary data is often serialized, meaning objects are converted into a format suitable for storage or transmission. [Bronson Zgeb demonstrates](https://www.google.com/url?sa=i&source=web&rct=j&url=https://bronsonzgeb.com/index.php/2021/08/28/save-data-with-binarywriter-and-binaryreader/&ved=2ahUKEwjitMb-rr-OAxW2UDABHetZDnsQy_kOegYIAwgAEC4&opi=89978449&cd&psig=AOvVaw1MFNboBH_gHJXWWkU2px3M&ust=1752686188450000)

Best practices for file handling

* Error Handling: Always wrap file operations in try-catch blocks to gracefully handle exceptions like FileNotFoundException, IOException, or DirectoryNotFoundException.
* Resource Management: Use using statements for StreamReader, StreamWriter, FileStream, BinaryReader, and BinaryWriter to ensure they are properly disposed of, freeing up file handles and system resources.
* Path Handling: Avoid hardcoding paths. [Medium advises](https://www.google.com/url?sa=i&source=web&rct=j&url=https://medium.com/@shehan.s/file-operations-in-c-net-part-01-file-and-directory-handling-60051437cc24&ved=2ahUKEwjitMb-rr-OAxW2UDABHetZDnsQy_kOegYIAwgAEDI&opi=89978449&cd&psig=AOvVaw1MFNboBH_gHJXWWkU2px3M&ust=1752686188450000) Use the Path class in System.IO to manipulate paths, combine path segments, and get file information in a platform-independent way. For instance, Path.Combine() is useful for constructing file paths.
* Choose the Right Tool: Select the appropriate class (File, StreamReader/StreamWriter, BinaryReader/BinaryWriter) based on the file size and type (text or binary).
* Permissions: Ensure your application has the necessary file system permissions to perform read/write operations. [Medium suggests](https://www.google.com/url?sa=i&source=web&rct=j&url=https://medium.com/@shehan.s/file-operations-in-c-net-part-01-file-and-directory-handling-60051437cc24&ved=2ahUKEwjitMb-rr-OAxW2UDABHetZDnsQy_kOegYIAwgAEDU&opi=89978449&cd&psig=AOvVaw1MFNboBH_gHJXWWkU2px3M&ust=1752686188450000)
* Asynchronous I/O: For performance-critical applications or when dealing with very large files, consider using asynchronous methods (e.g., ReadLineAsync(), WriteAsync()) to avoid blocking the main thread.

By following these guidelines and choosing the right classes, you can effectively manage file input and output in your C# applications. The next chapter will provide an example of building a simple console application using the concepts learned so far.

# 8. Putting it all together: Building a simple console application

Now that you've learned the fundamental concepts of C#, including syntax, variables, control flow, methods, OOP, and exception handling, it's time to apply this knowledge by building a practical console application. This chapter will guide you through creating a simple Task Manager Console Application.

This application will allow users to:

* Add new tasks.
* View all tasks.
* Mark tasks as complete.
* Exit the application.

The application uses a List<string> to store tasks and incorporates user input, conditional statements, loops, and basic exception handling.

Step-by-step example: Task Manager Console Application

## 1. Project setup

First, create a new C# Console Application project in Visual Studio or VS Code:

* Visual Studio: File > New > Project > Console App (.NET Core or .NET Framework) > Name your project (e.g., TaskManagerApp).
* VS Code: Open your project folder, then open the integrated terminal and run: dotnet new console -n TaskManagerApp followed by cd TaskManagerApp.

## 2. Design the Task class

A simple Task class will represent each task, including its description and completion status.

Create a new file named Task.cs (or simply add this class inside Program.cs for this small application).

csharp

// Task.cs (or within Program.cs)

public class Task

{

public string Description { get; set; }

public bool IsComplete { get; set; }

public Task(string description)

{

Description = description;

IsComplete = false; // New tasks are incomplete by default

}

// Method to display the task

public void DisplayTask()

{

string status = IsComplete ? "[X]" : "[ ]";

Console.WriteLine($"{status} {Description}");

}

}

* Description: A string to hold the task details.
* IsComplete: A bool to track if the task is finished.
* Constructor Task(string description): Initializes a new task.
* DisplayTask() method: Prints the task's status and description. The ternary operator condition ? value\_if\_true : value\_if\_false is used for concise conditional assignment.

## 3. Implement the main application logic (Program.cs)

Next, write the main program logic in Program.cs.

csharp

// Program.cs

using System;

using System.Collections.Generic; // Required for using List<T>

public class Program

{

// A list to store our tasks

private static List<Task> tasks = new List<Task>();

public static void Main(string[] args)

{

Console.WriteLine("Welcome to your Task Manager!");

RunTaskManager();

}

// Main loop for the task manager

private static void RunTaskManager()

{

bool running = true;

while (running)

{

DisplayMenu();

string choice = Console.ReadLine();

switch (choice)

{

case "1":

AddTask();

break;

case "2":

ViewTasks();

break;

case "3":

CompleteTask();

break;

case "4":

running = false; // Exit the loop

Console.WriteLine("Exiting Task Manager. Goodbye!");

break;

default:

Console.WriteLine("Invalid choice. Please try again.");

break;

}

Console.WriteLine("\nPress Enter to continue...");

Console.ReadLine(); // Pause until user presses Enter

Console.Clear(); // Clear console for next menu display

}

}

// Displays the main menu options

private static void DisplayMenu()

{

Console.WriteLine("\n--- Task Manager Menu ---");

Console.WriteLine("1. Add Task");

Console.WriteLine("2. View Tasks");

Console.WriteLine("3. Complete Task");

Console.WriteLine("4. Exit");

Console.Write("Enter your choice: ");

}

// Adds a new task to the list

private static void AddTask()

{

Console.Write("Enter task description: ");

string description = Console.ReadLine();

if (!string.IsNullOrWhiteSpace(description)) // Basic validation

{

tasks.Add(new Task(description));

Console.WriteLine("Task added successfully!");

}

else

{

Console.WriteLine("Task description cannot be empty.");

}

}

// Displays all tasks

private static void ViewTasks()

{

if (tasks.Count == 0)

{

Console.WriteLine("No tasks added yet.");

return;

}

Console.WriteLine("\n--- Your Tasks ---");

for (int i = 0; i < tasks.Count; i++)

{

Console.Write($"{i + 1}. "); // Display task number

tasks[i].DisplayTask();

}

}

// Marks a task as complete

private static void CompleteTask()

{

ViewTasks(); // Show tasks so user can pick one

if (tasks.Count == 0)

{

return; // No tasks to complete

}

Console.Write("Enter the number of the task to complete: ");

string input = Console.ReadLine();

// Basic error handling for user input

try

{

int taskNumber = int.Parse(input);

if (taskNumber > 0 && taskNumber <= tasks.Count)

{

tasks[taskNumber - 1].IsComplete = true; // Adjust for 0-based index

Console.WriteLine($"Task '{tasks[taskNumber - 1].Description}' marked as complete!");

}

else

{

Console.WriteLine("Invalid task number.");

}

}

catch (FormatException) // Catches if input is not a number

{

Console.WriteLine("Invalid input. Please enter a number.");

}

catch (Exception ex) // Catches any other unexpected errors

{

Console.WriteLine($"An error occurred: {ex.Message}");

}

}

}

## 4. Code explanation

* using System.Collections.Generic;: This statement imports the namespace containing the List<T> collection, which is used to store Task objects.
* private static List<Task> tasks = new List<Task>();: Declares a static list to hold Task objects. Being static means it belongs to the class itself, not to an instance, allowing all methods to access the same list of tasks.
* Main(string[] args): The program's entry point. It prints a welcome message and calls RunTaskManager().
* RunTaskManager(): This method contains the main application loop, controlled by a while loop that continues as long as running is true.
* DisplayMenu(): Presents the user with the available options.
* Console.ReadLine(): Reads input from the user.
* switch (choice): Handles the user's choice, calling the appropriate method.
* AddTask(): Prompts for a description, creates a new Task object, and adds it to the tasks list. It includes basic validation using string.IsNullOrWhiteSpace().
* ViewTasks(): Iterates through the tasks list using a for loop and calls the DisplayTask() method for each task. It handles the case where no tasks exist.
* CompleteTask():
  + First calls ViewTasks() to show the user the available tasks.
  + Uses int.Parse(input) to convert the user's string input into an integer.
  + Includes a try-catch block for exception handling:
    - FormatException: Catches if the user enters non-numeric input.
    - Exception: A general catch for any other unforeseen issues.
  + Validates the taskNumber to ensure it's within the valid range of tasks.
  + Marks the selected task as complete.
* Console.Clear(): Clears the console screen, providing a cleaner user interface between menu displays.

## 5. Running the application

* Visual Studio: Press F5 or click the "Start" button.
* VS Code / Terminal: Run dotnet run in the project directory.

You can now interact with your console-based Task Manager: add tasks, view them, mark them complete, and exit. This application demonstrates how various C# features combine to create a functional program.

This simple application can be extended in many ways, such as saving tasks to a file (using the File I/O concepts from the previous chapter), adding task editing functionality, or implementing more advanced features. The next chapter will provide guidance on continuing your C# learning journey.

# 9. Next steps and resources

Congratulations! You've successfully completed the foundational aspects of C# programming, from basic syntax to object-oriented principles, control flow, methods, and essential features like exception handling and file I/O. You've even built a simple application. This is a solid starting point, but the world of C# and .NET is vast and continually evolving.

This chapter outlines the next steps you can take to deepen your knowledge and provides resources for continued learning.

Learning beyond the basics

Here are some areas to explore as you advance your C# skills:

* Data Structures and Algorithms: Understanding how to efficiently organize and manipulate data is crucial for writing performant applications. Explore concepts like arrays, lists, queues, stacks, trees, and graphs, along with common algorithms for searching and sorting. C# provides built-in collections that implement these, like List<T>, Dictionary<TKey, TValue>, Stack, and Queue in the System.Collections.Generic namespace. [The DEV Community notes](https://www.google.com/url?sa=i&source=web&rct=j&url=https://dev.to/thedsdev/data-structures-in-c-a-beginners-guide-mastering-16ak&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAECE&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) that C# includes a built-in LinkedList class that allows dynamic memory allocation and easy insertion and deletion of nodes.
* Collections and LINQ: The .NET framework provides rich collections for storing data. Learn about List<T>, Dictionary<TKey, TValue>, HashSet<T>, and more. LINQ (Language Integrated Query) is a powerful feature that allows you to query and manipulate data from various sources (including collections, databases, XML) using a consistent syntax. You can use LINQ to filter, sort, group, and project data.
* Asynchronous Programming (Async/Await): For responsive applications, especially those dealing with I/O-bound operations (like network requests, database access, file I/O) or long-running CPU-bound tasks, mastering asynchronous programming with async and await is essential. These keywords simplify writing non-blocking code. [Auth0 explains](https://www.google.com/url?sa=i&source=web&rct=j&url=https://auth0.com/blog/introduction-to-async-programming-in-csharp/&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAECQ&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) that asynchronous programming executes code in a thread without waiting for I/O-bound or CPU-bound tasks to finish. [The NDepend Blog notes](https://www.google.com/url?sa=i&source=web&rct=j&url=https://blog.ndepend.com/c-async-await-explained/&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAECU&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) that the C# compiler transforms the use of await into a state machine that manages suspending and resuming execution. [Microsoft Learn advises](https://www.google.com/url?sa=i&source=web&rct=j&url=https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAECY&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) that an async method should include one or more await expressions; otherwise it fails to yield, leading to inefficient code, even though it compiles.
* Delegates, Events, and Lambda Expressions: These are advanced language features used for event handling, callbacks, and functional programming patterns.
* Generics: Understanding and using generics allows you to write reusable code that works with different data types without sacrificing type safety.
* Web Development (ASP.NET Core / Blazor): If you're interested in building web applications, delve into ASP.NET Core MVC (Model-View-Controller) or the newer Blazor framework. [Visual Studio Magazine says](https://www.google.com/url?sa=i&source=web&rct=j&url=https://visualstudiomagazine.com/articles/2024/08/07/integrating-blazor-with-existing-,-d-,net-web-apps.aspx&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAECo&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) Blazor enables you to build interactive web UIs using C# instead of JavaScript, leveraging WebAssembly for client-side execution. [Learn Microsoft notes](https://www.google.com/url?sa=i&source=web&rct=j&url=https://learn.microsoft.com/en-us/aspnet/core/blazor/?view%3Daspnetcore-9.0&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAECs&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) that Blazor Web Apps can quickly deliver UI to the browser by statically rendering HTML content from the server in response to requests, and further improve user experience with enhanced navigation and streaming rendering.
* Game Development (Unity): C# is the primary scripting language for the Unity game engine. If game development appeals to you, start learning Unity and applying your C# knowledge there. [Unity mentions](https://www.google.com/url?sa=i&source=web&rct=j&url=https://unity.com/how-to/programming-unity&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAEC0&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) that you can use C# scripts to develop pretty much every part of a game or other real-time interactive content, and that Unity supports scripting in C# with two main architectural approaches: object-oriented design and data-oriented design, the latter using its Data-Oriented Technology Stack (DOTS) for high performance and multithreading.
* Desktop Applications (WPF, WinForms): While web and mobile development are popular, C# is still widely used for traditional desktop applications using frameworks like WPF (Windows Presentation Foundation) or WinForms.
* Databases (Entity Framework Core): Learn how to interact with databases using Entity Framework Core (EF Core), Microsoft's object-relational mapper (ORM) for .NET. [Microsoft Learn provides documentation](https://www.google.com/url?sa=i&source=web&rct=j&url=https://learn.microsoft.com/en-us/ef/dotnet-data/&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAEDA&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) on how to work with data using .NET (C#), offering guides and tutorials for various databases and data access approaches.
* Testing (Unit Testing): Learn to write unit tests for your C# code using frameworks like NUnit or xUnit. Effective testing is critical for building reliable software. [MentorCruise recommends](https://www.google.com/url?sa=i&source=web&rct=j&url=https://mentorcruise.com/books/csharp/&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAEDI&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000" \t "_blank) "Unit Testing Principles, Practices, and Patterns" by Vladimir Khorikov for learning to design and write tests that target key areas of your code.

## Recommended learning resources

* Official Microsoft Documentation: [Learn Microsoft offers comprehensive C# documentation](https://www.google.com/url?sa=i&source=web&rct=j&url=https://learn.microsoft.com/en-us/dotnet/csharp/&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAEDQ&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) which includes guides, tutorials, and code samples, and is always up-to-date with the latest C# and .NET features.
* Online Courses: Platforms like [Udemy](https://www.google.com/url?sa=i&source=web&rct=j&url=https://www.udemy.com/course/advanced-topics-csharp/&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAEDY&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) and [MentorCruise](https://www.google.com/url?sa=i&source=web&rct=j&url=https://mentorcruise.com/courses/csharp/&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAEDc&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000" \t "_blank) offer structured C# courses, from beginner to advanced. [Quora notes](https://www.google.com/url?sa=i&source=web&rct=j&url=https://www.quora.com/What-is-the-best-resource-to-learn-C&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAEDg&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) that Programming with Mosh on Udemy is popular for learning C#.
* Books: For deeper dives, consider books like "C# in Depth" by Jon Skeet or "The C# Player's Guide" by RB Whitaker. [MentorCruise recommends](https://www.google.com/url?sa=i&source=web&rct=j&url=https://mentorcruise.com/books/csharp/&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAEDo&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000" \t "_blank) several books, including "C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals" for a solid foundation in building projects.
* Coding Practice Platforms: Websites like LeetCode, HackerRank, and CodeSignal offer coding challenges to sharpen your problem-solving and algorithmic skills.
* Project-Based Learning: Start building small projects based on your interests. For instance, if you're interested in game development, try building a simple Tic-Tac-Toe game. [InterviewBit mentions](https://www.google.com/url?sa=i&source=web&rct=j&url=https://www.interviewbit.com/blog/c-sharp-projects/&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAED0&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000" \t "_blank) a note-taking application as among the best C# projects for beginners, simple to complete and illustrating various C# concepts. [Uplyrn suggests](https://www.google.com/url?sa=i&source=web&rct=j&url=https://uplyrn.com/post-details/20-c-sharp-project-ideas-to-sharpen-up-your-programming-skills&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAED4&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000" \t "_blank) building an e-commerce web application, a Twitter bot, or an ebook library and reader, while [InterviewBit suggests](https://www.google.com/url?sa=i&source=web&rct=j&url=https://www.interviewbit.com/blog/c-sharp-projects/&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAED8&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000" \t "_blank) for intermediate developers, projects such as a chat application, a car racing game, a music player, or an e-commerce website.
* IDEs & Tools: Become proficient with Visual Studio (the full IDE) or Visual Studio Code (a lightweight editor with C# extensions).
* Community & Support: Engage with the C# developer community on platforms like [Stack Overflow](https://www.google.com/url?sa=i&source=web&rct=j&url=https://stackoverflow.com/questions/26939468/how-to-run-a-linq-query-on-a-collection-of-objects-and-a-collection-inside-of-ea&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAEEI&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000), Reddit (e.g., r/csharp), and Discord/Slack groups. Quora recommends reading articles, commenting opinions, joining forums and GitHub, following experts on Twitter, and attending conferences.

## Continuing your journey

Remember that consistency and practice are key to becoming proficient in C#. Don't be afraid to experiment, make mistakes, and seek help from the community. With the foundations you've built and the resources available, you're well-equipped to tackle more complex challenges and become a skilled C# developer. [Quora advises](https://www.google.com/url?sa=i&source=web&rct=j&url=https://www.quora.com/What-are-some-good-resources-to-learn-C-from-scratch&ved=2ahUKEwio4Pusr7-OAxWxrokEHd2vNRQQy_kOegYIAwgAEEQ&opi=89978449&cd&psig=AOvVaw2pv2cJ0_-nem7e0ei4OlO5&ust=1752686285808000) to know your pace and take breaks when learning C# programming. Keep coding!