

YUKO TAKAHASHI

ESSENTIAL

C#

**KNOWLEDGE
FOR
BEGINNERS:
BEST 100**

2024 Edition



JUST READ THIS
AND YOU'RE SET!

Index

[Static Typing](#)
[Value Types and Reference Types](#)
[Properties in C#](#)
[Indexers in C#](#)
[Events in C#](#)
[Delegates in C#](#)
[Lambda Expressions in C#](#)
[LINQ \(Language Integrated Query\) in C#](#)
[Nullable Types in C#](#)
[Async/Await in C#](#)
[Exception Handling in C#](#)
[Attributes in C#](#)
[Reflection in C#](#)
[Generics in C#](#)
[Extension Methods](#)
[Partial Classes and Methods](#)
[Anonymous Types in C#](#)
[Dynamic Types in C#](#)
[Checked and Unchecked](#)
[Iterators](#)
[nameof Operator](#)
[Null Conditional Operator](#)
[String Interpolation in C#](#)
[Pattern Matching in C#](#)
[Local Functions in C#](#)
[Tuples in C#](#)
[Discards in C#](#)
[Ref Locals and Returns](#)
[Out Variables](#)
[In Parameter Modifier](#)
[Readonly Members in C#](#)
[Default Interface Methods in C#](#)
[Using Declarations](#)
[Switch Expressions](#)
[C# Records](#)
[Init Only Setters](#)
[Top-level Statements](#)
[Global Using Directives](#)
[File-scoped Namespace Declaration](#)
[Nullable Reference Types](#)
[C# 10.0 - Record structs](#)
[C# 10.0 - Extended Property Patterns](#)

[Natural Type Expressions in C# 10.0](#)
[Global Using Directives in C# 10.0](#)
[File-scoped Namespace Enhancement in C# 10.0](#)
[List Patterns in C# 11.0](#)
[C# 11.0 - Required Properties](#)
[C# 11.0 - Raw String Literals](#)
[UTF-8 String Literals in C# 11.0](#)
[Enhanced #line Directive in C# 11.0](#)
[C# Compiler \(Roslyn\)](#)
[.NET Runtime](#)
[Entity Framework Basics](#)
[Introduction to ASP.NET Core](#)
[Xamarin for Mobile Development](#)
[Visual Studio](#)
[Visual Studio Code for C#](#)
[NuGet](#)
[Understanding MSBuild](#)
[Unit Testing in C# with NUnit and xUnit](#)
[Design Patterns in C#](#)
[SOLID Principles in C#](#)
[Dependency Injection in C#](#)
[Concurrency and Multithreading in C#](#)
[Memory Management in C#](#)
[Performance Optimization in C#](#)
[Security Practices in C#](#)
[Cross-Platform Development with .NET Core](#)
[Code Analysis Tools in C#](#)
[Application Lifecycle Management with C#](#)
[Source Control Integration in C#](#)
[Continuous Integration/Continuous Deployment \(CI/CD\) with C#](#)
[Database Connectivity in C#](#)
[API Development in C#](#)
[Cloud Services Integration in C#](#)
[Windows Presentation Foundation \(WPF\)](#)
[Windows Forms in C#](#)
[Blazor for Web Development](#)
[Universal Windows Platform \(UWP\) Basics](#)
[Exploring C# Interactive \(CSI\)](#)
[REPL in Visual Studio](#)
[Code Refactoring Tools in Visual Studio](#)
[Static Code Analysis in C#](#)
[Code Profiling in C#](#)
[Code Documentation in C#](#)
[Assembly Versioning in C#](#)

[Localization and Globalization in C#](#)

[Data Types and Variables in C#](#)

[Control Structures in C#](#)

[Object-Oriented Programming in C#](#)

[Interfaces in C#](#)

[Events and Delegates in C#](#)

[File I/O in C#](#)

[Error Handling in C#](#)

[Data Access in C#](#)

[Web Development with C#](#)

[Mobile Development with C#](#)

[Game Development with Unity and C#](#)

[IoT Development with C#](#)

[Machine Learning with ML.NET](#)

[True and False Values in C#](#)

[Boolean Logic in C#](#)

[Nullable Boolean Types in C#](#)

[Truth Tables in C#](#)

Introduction



Welcome to a tailored learning journey in the world of C# programming. Designed for individuals who already grasp basic programming concepts, this book aims to equip beginners with the essential knowledge necessary to master C#. Each section of this guide is crafted to ensure that you gain a deep understanding of key C# elements without overwhelming details.

Whether you're starting out or revisiting the fundamentals as a seasoned programmer, this book serves as a focused resource to brush up on the essentials. Our concise approach allows you to efficiently learn and apply your skills in practical scenarios.

We encourage you to leave a review or comment after your reading experience. Your feedback not only helps us improve but also assists your fellow engineers in discovering this valuable resource. Sharing your thoughts and insights can greatly benefit others in similar positions, fostering a community of learning and growth.

Static Typing

Static typing in C# means that variable types are explicitly declared and determined at compile time.

In the following example, we assign integers and strings to variables, demonstrating C#'s static typing.

[Code]

```
int number = 5;  
string greeting = "Hello, world!";  
Console.WriteLine(number);  
Console.WriteLine(greeting);
```

[Result]

```
5  
Hello, world!
```

In this example, the variable `number` is explicitly declared as an `int`, and `greeting` is declared as a `string`. This is essential in C# because the type of each variable is fixed at compile time and cannot change throughout the program, which helps prevent many common type-related errors that can occur in dynamically typed languages. The explicit type declaration enhances code readability, debugging, and performance optimization, as the compiler can make more assumptions and optimizations.

[Trivia]

Static typing helps catch errors at compile time rather than at runtime, which generally results in more robust and maintainable code. It also allows Integrated Development Environments (IDEs) to provide features like type inference, code completion, and more effective refactoring tools.

2

Value Types and Reference Types

C# distinguishes between value types and reference types, which are stored and handled differently in memory.

Below is an example showcasing the difference between a value type and a reference type.

[Code]

```
int value1 = 10;
int value2 = value1;
value2 = 20;
Console.WriteLine("Value1: " + value1); // Output will be based on the value type behavior
Console.WriteLine("Value2: " + value2); // Demonstrates independent copy behavior
string ref1 = "Hello";
string ref2 = ref1;
ref2 = "World";
Console.WriteLine("Ref1: " + ref1); // Output will reflect reference type behavior
Console.WriteLine("Ref2: " + ref2); // Shows independent reference due to string immutability
```

[Result]

```
Value1: 10
Value2: 20
Ref1: Hello
Ref2: World
```

In the value type example (int), changing value2 does not affect value1 because when value 2 is assigned value1, a new independent copy of the value is created. In contrast, with reference types (string in this case), both ref1 and ref2 initially point to the same data. However, strings are immutable in C#, so when ref2 is changed, it actually points to a new string object, leaving ref1 unchanged. This behavior is crucial for understanding how memory management and data manipulation work in C#, affecting performance and functionality.

[Trivia]

Understanding the distinction between value types and reference types is essential for managing memory efficiently in C#. Value types are stored on the stack, which allows quicker access but limited flexibility. Reference types are stored on the heap, which is more flexible but requires overhead for memory management.⁴

Properties in C#

Properties in C# are members that provide a flexible mechanism to read, write, or compute the values of private fields.

The following example illustrates a simple class with a private field and a property that provides access to this field.

[Code]

```
public class Person
{
    private string name; // Private field
    // Public property
    public string Name
    {
        get { return name; } // Get method
        set { name = value; } // Set method
    }
}
class Program
{
    static void Main()
    {
        Person person = new Person();
        person.Name = "Alice"; // Using the set accessor
        Console.WriteLine(person.Name); // Using the get accessor
    }
}
```

[Result]

Alice

In C#, properties play a crucial role in encapsulation, one of the fundamental principles of object-oriented programming. They allow the class to control how important fields are accessed and modified. In our example: `private string name;` This line declares a private field named `name`. This field cannot be accessed directly from outside the class, which protects the field from unwanted external modifications. `public string Name:` This property acts as a safe way to

access the private field. It includes two parts:
`get { return name; }`: This is a get accessor, used to return the value of the private field. When someone outside the class wants to get the value of `Name`, this accessor is invoked.
`set { name = value; }`: This is a set accessor, used to assign a new value to the private field. The `value` keyword represents the value being assigned to the property. Properties can also be read-only (if they have no set accessor) or write-only (if they have no get accessor), depending on the needs of your program.

[Trivia]

In more advanced scenarios, properties can use more complex logic in get and set accessors, not just simple assignments. For example, you could add validation in the set accessor to check the incoming value before setting the field.

4

Indexers in C#

Indexers allow instances of a class or struct to be indexed just like arrays.

The following example demonstrates a class that simulates an internal array through an indexer.

[Code]

```
public class SimpleArray
{
    private int[] array = new int[10]; // Internal private array
    // Indexer definition
    public int this[int index]
    {
        get { return array[index]; } // Get indexer
        set { array[index] = value; } // Set indexer
    }
}
class Program
{
    static void Main()
    {
        SimpleArray sa = new SimpleArray();
        sa[0] = 10; // Using the set indexer
        sa[1] = 20; // Using the set indexer
        Console.WriteLine(sa[0]); // Using the get indexer
        Console.WriteLine(sa[1]); // Using the get indexer
    }
}
```

[Result]

```
10
20
```

Indexers in C# are a syntactic convenience that allows an object to be indexed like an array. Here's a deeper look at the indexer used in the example: `private int[] array = new int[10];`; Thi

s private field holds an array of integers. The array is used internally to store data.
`public int this[int index];` This is the syntax for declaring an indexer. The `this` keyword is followed by a parameter list enclosed in square brackets. The parameter indicates the index position.
`get { return array[index]; }` The get accessor for the indexer returns the value at the specified index.
`set { array[index] = value; }` The set accessor assigns a value to the array at the specified index. The `value` keyword represents the value being assigned. Indexers can be very powerful, especially in classes that encapsulate complex data structures. They provide a way to access elements in a class that encapsulates a collection or array with the simplicity of using array syntax.

[Trivia]

You can define indexers not only for single parameters but also for multiple parameters, enhancing the capability to mimic multi-dimensional arrays or more complex data structures.⁴

5

Events in C#

Events in C# are a way to send notifications to multiple subscribers when something significant happens in your program.

Below is a simple example demonstrating how to declare and use events in C#. We'll create a class that triggers an event when a method is called.

[Code]

```
using System;
public class ProcessBusinessLogic
{
    // Declare the delegate
    public delegate void ProcessCompletedEventHandler(object sender, EventArgs e);
    // Declare the event
    public event ProcessCompletedEventHandler ProcessCompleted;
    // Method to trigger the event
    public void StartProcess()
    {
        Console.WriteLine("Process Started.");
        // Some process logic here
        OnProcessCompleted(EventArgs.Empty);
    }
    protected virtual void OnProcessCompleted(EventArgs e)
    {
        // Check if there are any subscribers
        ProcessCompleted?.Invoke(this, e);
    }
}
class Program
{
    static void Main(string[] args)
    {
        ProcessBusinessLogic pbl = new ProcessBusinessLogic();
        pbl.ProcessCompleted += (sender, e) => Console.WriteLine("Process Completed.");
        pbl.StartProcess();
    }
}
```

[Result]

Process Started.
Process Completed.

In the provided example, we have a class called `ProcessBusinessLogic`. This class contains a delegate named `ProcessCompletedEventHandler`, which defines the signature for methods that can respond to the event. The event itself, `ProcessCompleted`, uses this delegate type. The method `StartProcess` is used to simulate a process. When this method is called, it eventually triggers the `ProcessCompleted` event by calling `OnProcessCompleted`, which checks if there are any subscribers to the event. If there are, it invokes the event, passing itself (this) and an `EventArgs` object to the subscribers. This pattern allows other parts of your application to react to changes or significant actions within the class without tightly coupling the components.

[Trivia]

Events in C# are built on the delegate model. An event can have multiple subscribers, and when the event is triggered, all subscribers are notified in the order they subscribed. This is particularly useful in designing loosely coupled systems and is a key aspect of the observer design pattern.

6

Delegates in C#

Delegates are type-safe function pointers in C# that allow methods to be passed as parameters.

Below is a basic example showing how to declare, instantiate, and use delegates in C#. We'll create a delegate that points to a method performing a simple calculation.

[Code]

```
using System;
public delegate int CalculationHandler(int x, int y);
class Program
{
    static int Add(int a, int b)
    {
        return a + b;
    }
    static void Main(string[] args)
    {
        CalculationHandler handler = Add;
        int result = handler(5, 6);
        Console.WriteLine("Result of addition: " + result);
    }
}
```

[Result]

Result of addition: 11

In the example, a delegate named `CalculationHandler` is defined that can reference any method that takes two integers as parameters and returns an integer. In this case, the delegate references the `Add` method. The `Main` method creates an instance of `CalculationHandler`, pointing it to the `Add` method. When `handler` is invoked with two integers, it effectively calls `Add` with those integers and outputs the result. This demonstrates how delegates are used to encapsulate a method reference and pass it around like any other variable or parameter. This is instrumental in creating flexible and reusable components such as event handling systems or callback mechanisms.

[Trivia]

Delegates are the foundation of many advanced .NET framework features such as events and LINQ. They are integral to understanding asynchronous programming patterns in C#, including the use of `async` and `await` keywords.⁴

7

Lambda Expressions in C#

Lambda expressions in C# provide a concise way to write inline expressions or functions that can be used to create delegates or expression tree types.

Here is a simple example of a lambda expression used to filter a list of integers.

[Code]

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
Func<int, bool> isEven = x => x % 2 == 0;
List<int> evenNumbers = numbers.Where(isEven).ToList();
foreach (var num in evenNumbers)
{
    Console.WriteLine(num);
}
```

[Result]

```
2
4
```

In this code, `List<int> numbers` initializes a list of integers. The lambda expression `x => x % 2 == 0` defines a function that checks if a number is even. This function is used as a delegate `Func<int, bool>`, where `int` is the input type and `bool` is the return type, indicating the result of the condition. The method `Where(isEven)` is a LINQ method that filters the list based on the lambda function. Only numbers satisfying the condition (even numbers) are included in the new list `evenNumbers`, which is materialized into a list with `.ToList()`. The `foreach` loop then iterates over `evenNumbers` and prints each element. Lambda expressions are essential for writing concise and readable code, particularly when working with data manipulation and LINQ.

[Trivia]

Lambda expressions in C# are derived from lambda calculus, which is a framework developed in the 1930s to study functions and their application. In programming, they provide a powerful tool for creating concise and flexible functional-style code.

LINQ (Language Integrated Query) in C#

LINQ is a powerful feature in C# that allows developers to query various data sources (like arrays, enumerable classes, XML, relational databases) in a consistent manner.

Below is an example of using LINQ to query an array of names to find names that contain the letter 'a'.

[Code]

```
string[] names = { "Alice", "Bob", "Carol", "David" };
var namesWithA = from name in names
                  where name.Contains('a')
                  select name;
foreach (var name in namesWithA)
{
    Console.WriteLine(name);
}
```

[Result]

```
Alice
Carol
David
```

This example initializes an array `string[] names` with several names. The LINQ query is written using query syntax. It starts with the `from` clause, which specifies the data source (`names`). The `where` clause filters names containing the letter 'a'. The `select` statement specifies that these names should be selected into the resulting sequence `namesWithA`. The query syntax in LINQ closely resembles SQL (Structured Query Language), making it intuitive for those familiar with database querying. The result is executed lazily; the actual query execution occurs at the `foreach` loop, where each filtered name is printed. LINQ simplifies data querying by integrating query capabilities directly into the C# language, allowing for more readable and maintainable code.

[Trivia]