C# Mini Reference 2023

A Quick Guide to the Modern C# Programming Language for Busy Coders

Harry Yoon

Version 1.1.3, 2023-04-04

Copyright

C# Mini Reference:
A Quick Guide to the C# Programming Language

© 2023 Coding Books Press

All rights reserved. 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 embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Published: January 2023

Harry Yoon
San Diego, California

ISBN: 9798372476691

Preface

C# was originally created based on the Java programming language, circa 2000~2002, partly because Microsoft wanted to lure programmers who were familiar with Java at the time.

Java had been created a few years earlier when the object-oriented programming (OOP) was becoming the most dominant programming paradigm in the industry. Java became wildly popular. In Java, *everything was a class*. Functions were demoted to the "methods" of a class. Even the functions that do not really belong to an object needed to be part of a class, e.g., as "static" methods. You couldn’t program in Java without creating a class first, at least syntactically.

C# initially followed this "everything is a class" design. Now, for the past 20+ years, C# has been trying to liberate itself from the ghost of Java.

In recent years, people have been realizing the limitations of the OOP, especially as a one-size-fits-all programming paradigm. Most modern programming languages like Go, Rust, Swift, and even other JVM languages like Scala and Kotlin do not impose such constraints. In fact, Java and C# are the only two widely used languages that have such built-in class-first constraints.

C# has been constantly evolving. But, it went through major changes between C# 6 and C# 7. And, ever since. Now, C# releases a new major version every year. As of this writing, C# 11 is the most current version.

Although there have been many (small and big) updates over the years, the recent changes are really about C# becoming free of Java. C# is now officially a "multi-paradigm" programming language, and not just an OOP language. The modern C# encourages the functional programming styles, among other things. For example, a lot of constructs in C# are now expressions rather than statements. (Also, it should be noted that many new features of C# are borrowed from F#, another .NET programming language, which is primarily functional.) Since C# 9, we can use the top-level statements, at least in no more than one source code file in a program. This kind of (cosmetic) changes allow us to do away with much of the boilerplate code in the modern C#.

C# 11 now supports so-called "static abstract methods" in interfaces. Although its full implications are not entirely clear, this change makes C# more on a par with other (more modern) languages. C# 11's interface is comparable, in terms of its capabilities, to Go's interface, Rust's trait, and Swift's protocol, among others.

This book primarily focuses on the syntax of the C# language. But we encourage the readers to read beyond the syntax. In particular, we will need to adjust our programming styles, rather than sticking with the old "everything is a class" paradigm, to take advantage of the new modern features and to program more effectively in C#.

Having said that, you can read the book more or less from beginning to end, if you'd like, and you will get the full picture of the C# language.

One thing to note is that there's a fair amount of cross references, and if you have no prior experience with programming in C#, or other similar languages like C++ or Java, you may have a little difficulty following some parts of the book. We also skip some basics of C# so that we can cover more modern features in more detail in the limited space. The book's intended audience is advanced beginners to intermediate-level programmers. Advanced C# developers can also benefit from this book by selectively going through more advanced, and modern, features.

It should be noted that this book is not an authoritative language reference. If there are any inconsistencies or confusing, or seemingly incorrect, explanations in the book, we recommend the readers to refer to the official language specification.

Dear Readers:

Please read b4 you purchase, or start investing your time on, this book.

A programming language is like a set of standard lego blocks. There are small ones and there are big ones. Some blocks are straight and some are L-shaped. You use these lego blocks to build spaceships or submarines or amusement parks. Likewise, you build programs by assembling these building blocks of a given programming language.

This book is a *language reference*, written in an informal style. It goes through each of these lego blocks, if you will. This book, however, does not teach you how to build a space shuttle or a sail boat. If this distinction is not clear to you, it’s unlikely that you will benefit much from this book. This kind of language reference books that go through the syntax and semantics of the programming language broadly, but not necessarily in gory details, can be rather useful to programmers with a wide range of background and across different skill levels.

This book is not for complete beginners, however. When you start learning a foreign language, for instance, you do not start from the grammar. Likewise, this book will not be very useful to people who have little experience in real programming. On the other hand, if you have some experience programming in other languages, and if you want to quickly learn the essential elements of this particular language, then this book can suit your needs rather well.

Ultimately, only you can decide whether this book will be useful for you. But, as stated, this book is written for a wide audience, from beginner to intermediate. Even experienced programmers can benefit, e.g., by quickly going through books like this once in a while. We all tend to forget things, and a quick regular refresher is always a good idea. You will learn, or re-learn, something "new" every time.

Good luck!

Table of Contents

Chapter 1. Introduction

C# is a rather complex language. In fact, depending on how you measure the complexity of a programming language, C# could be the most complex language among some of the widely used languages. No one likes complexity. And yet, C# is one of the most popular and favorite languages for many developers. *Why?*

A programming language is a tool. Sometimes, having a complex and flexible tool can make doing a real task much simpler and easier. Using a chainsaw can make the task of cutting a tree easier, for example. That’s the case with C#.

There are a few implications for this. On the one hand, C# is not generally considered a beginner-friendly language. There is an upfront investment you need to make before you can reap the benefits. On the other hand, C# can be a powerful tool for building a large, enterprise-grade software once you master the tool.

In general, you will end up with "simpler" programs using "more complex" languages like C#. At least, that’s the idea. Hence, it is rather important to use C# *correctly* and *effectively*. Otherwise, the downside of using a heavyweight language, if you will, will be bigger than the benefits of using such a language.

This book provides a bird’s eye view of the C# language grammar. It is organized into a few dozen semi-independent chapters, covering most of the important features of C#.

A programming language is not just a sum of its features. In order to be able to use a language like C# most effectively, however, you will need to know what features are available at your disposal. You will need to have a thorough high-level understanding of each of these features. As stated in the preface, the readers are encouraged not to lose the sight of the forest while going through each of these trees.

1.1. C# and .NET

At the risk of oversimplifying, there are three aspects to a programming language, and programming in that language:

- The language itself, e.g., the grammar and the standard library,
- The runtime, and
- The tooling (e.g., the language compiler).

As an example, the C++ programming language specification is managed by the ISO. On the other hand, toolings are provided by various tool vendors like Microsoft, GNU/gcc, and clang. C++ has no separate runtime per se, other than the standard library support at run time. C++ programs are directly compiled to a machine code.

In case of C# and .NET, Microsoft more or less owns a whole ecosystem comprising all three parts of the C# language. (We do not discuss other implementations of C# such as those used in Unity, etc.) The .NET system defines a set of standard libraries and their run time support. .NET is also a runtime, or a virtual machine, that runs the byte code compiled from C# programs.

Microsoft's .NET has a very complicated history, with many different versions and manifestations, and with many different (and ever-changing) names. The framework, and the version, that we use in this book is simply called .NET 7 (which was originally called .NET Core). This is a cross-platform version of the .NET framework. That is, once you create a C# program targeting .NET 7, you can run it on any system that is supported by .NET 7. (In addition, .NET, in general, supports multiple programming languages.)

Another implication of this rather tightly integrated ecosystem is that C# does not stand by itself. In fact, at least in principle, you will not be able to read and interpret C# programs precisely without knowing what their associated .NET framework is.

This is not necessarily unique to C#, with all programming languages evolving over time, etc. But, in case of C# and .NET, the interdependency is much tighter.

C# uses a project file to manage build configurations and what not. The default version of C# for .NET 7 is C# 11. That is, if you plan to use a different version of C# on .NET 7, for instance, then you will need to explicitly specify that in the project file.

When you scaffold an empty C# project on .NET 7, e.g., using Visual Studio or dotnet CLI, it creates the following placeholder project file:

SampleProject.csproj

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

There are four properties set by the tool in this file. The `OutputType` property indicates the type of the target binary, e.g., Exe (executable) vs Lib (library). We are targeting .NET 7 via `TargetFramework` (which is a natural value if you use the tooling based on .NET 7).

The last two properties, `ImplicitUsings` and `Nullable`, are rather significant. C# went through a few backward incompatible changes in the past. In the original C#, the values of both these properties would have been `disable`. Now, moving forward, the majority of the C# community will likely use the `enable` value, and hence, C# has become effectively backward incompatible.

We will always assume in this book that these two values are set to `enable`. Strictly speaking, some of the information provided in this book might be "incorrect", e.g., when using different values for these properties (and, possibly for some others as well).

Unless you have a legacy code, etc., we recommend the readers do the same: Always use `<ImplicitUsings>enable</ImplicitUsings>` and `<Nullable>enable</Nullable>`. `ImplicitUsings` has a relatively small impact, but `Nullable` completely changes how C# program works. We discuss this in the next section.

1.2. Nullable Context

A variable of a reference type can be `null`. The null value simply means that the variable points to nothing. That it points to no real data in memory. The default value of a reference variable is `null`.

This is true across all different C-style programming languages which support the reference types or pointer types. Accessing these variables when they are `null` has been the cause of so many (potentially preventable) errors, called the null-pointer exceptions.

C# has a *compile time* support to address this problem. This feature is called the "nullable context". It is `disabled` by default, which means that all reference variables are nullable, as they always have been. In the `enabled` nullable context, the traditional (nullable) reference variables become non-nullable. Setting these variables to `null` will cause a compile time error. In addition, one can use a new kind of reference variables within the `enabled` nullable context, called the "nullable reference variables". All nullable reference variables should be checked for nullability before using. Otherwise, the compiler will likely throw compile-time errors or warnings.

You can set the nullable context in the C# project file as we discussed in the previous section. In addition, you can selectively enable or disable the nullable context in certain parts of the code using the C#'s compiler directive, `nullable`. To enable, you can put this in your source code file:

```
#nullable enable
```

This is in effect until the end of the source file, or until the next `#nullable` directive is used, if any. To disable, use the following:

```
#nullable disable
```

You can also restore the previous context value using the following directive:

```
#nullable restore
```

The nullable context is purely a compile time support. The .NET runtime does not know anything about the nullable context, and as far as it is concerned, all reference variables are nullable.

Here’s a list of valid values for the `nullable` setting:

`disable`	Sets the nullable annotation and warning contexts to disabled
`enable`	Sets the nullable annotation and warning contexts to enabled
`restore`	Restores the nullable annotation and warning contexts to project settings

`disable annotations`	Sets the nullable annotation context to disabled
`enable annotations`	Sets the nullable annotation context to enabled
`restore annotations`	Restores the nullable annotation context to project settings
`disable warnings`	Sets the nullable warning context to disabled
`enable warnings`	Sets the nullable warning context to enabled
`restore warnings`	Restores the nullable warning context to project settings

1.3. Book Organization

We start from the top, in terms of the general C# program structure. A C# program is essentially a collection of C# source files that are (to be) compiled together. An executable C# program has one and only entry point, which could be a `Main` method of a startup class or the top-level statements in one source code file. We also describe the basic concepts of access control, e.g., `public` vs `private`, namespaces, and blocks and scopes in this chapter.

A source code file is essentially a sequence of characters. We next go through some basic process of converting an input sequence of characters to a sequence of "tokens" which the compiler understands. This is generally known as "lexing". C#'s lexical structure is rather similar to those of other C-style programming languages.

A C# program logically consists of one or more namespaces. A namespace can include other namespaces or declarations of custom types, e.g., using `class`, `record`, and `struct`, etc., which can in turn include other member declarations. In C#, the `using` directives are used to import names from other namespaces. Namespaces themselves do not participate in access control.

C# types are organized in a hierarchy with the base class `object` at its root. As with many C-style languages, C# has values and references (or, pointers), with different characteristics. C# supports two distinct, and mutually exclusive, categories of types, namely, value types and reference types. In addition, C# includes the `dynamic` type, which is sort of an escape hatch when static typing is too limiting in certain cases.

In the next two chapters, Constants and Variables, we go through some basics of "variables" in C#. Variables are an essential component of imperative programming, in which we manipulate the state (variables) to perform the desired task. In C#, there are many different categories of variables. Constants are a special kind of variables whose values are known, and fixed, at compile time. Non-constant variables in C# can be classified into multiple categories, including local variables, fields of a class (or, record or struct), etc. Some fields belong to a class itself ("static fields"), and some fields to an object, or an instance of a class ("instance fields").

Local variables are the most common kind of variables, e.g., in (non-OOP) C-style programming languages. They are tied to a call stack frame, and when the stack frame is removed, their memory is de-allocated. (And/or, the memory pointed by these variables is marked for garbage collection.)

C# does not support standalone functions (that are not a member of a class), as we alluded in the preface. This is a fundamental limitation of the languages like Java and C#. To compensate this syntactic limitation, C# includes a number of different function kinds. Local functions and static local functions are two of those function kinds. Similar to local variables, local functions are tied to a stack frame, and they have a limited lifetime. Local functions in C# are similar to the lambda functions, and they have overlapping use cases.

C# includes many "function-like" constructs, including local functions, function members (or, methods), static or instance constructors, and delegates, etc. Their declarations and invocations have more or less common syntax. In the Formal Parameters chapter, we describe the common structures of their "function parameter" declarations.

Function arguments are by default "passed by value" (which can have slightly different meanings when applied to values vs references). C# allows using `in`, `out`, and `ref` parameter modifiers to change this argument passing semantics in some cases.

We next go through C#'s built-in value types such as `bool`, `char`, and other numeric types. We include brief descriptions of these types for completeness.

Continuing with the builtin types, we go over some basics of strings in the next chapter, Strings. String is a reference type, but it has characteristics of value types, e.g., in terms of value equality, and the like. In this chapter, we primarily focus on a number of different string literal syntax in C#. But, readers are encouraged to learn more about strings through different resources. For example, C# strings have many (important) builtin methods, which we do not cover in this book.

Generics is a very important part of C#. Types, and methods, can be declared with "type parameters", which in effect allows us to define a set of related types in one declaration. The most important recent developments in C# with respect to generics have been improvements in generic type constraints. C# now allows constraining type parameters in generic declarations in many different and flexible ways.

An interface in the modern C# defines a behavior, and a lot more. In fact, `interface` has changed so much recently that it is now hard to tell what an interface really is in C#.

Among the many (important) recent changes, there are a few that's noteworthy. First, methods in an `interface` can now have default implementations, which can be inherited by an implementing class (and, record and struct). Second, static methods have become an integral part of `interface`, and not just supporting actors, if you will. Third, static methods can now even participate in defining the "behavior" part of an interface, as of C# 11. For example, C# 11's new generic math support includes the `INumber` interface in the `Systems.Numerics` namespace:

```
public interface INumber<TSelf> : IComparable<TSelf>, ...
where TSelf : INumber<TSelf> {
    public static virtual int Sign (TSelf value);
    // ...
}
```

All types implementing `INumber<T>` will need to implement this `Sign` virtual static method, for instance. (This partial example also illustrate the use of generic type constraints.) In the Interfaces chapter, we go through some essential elements of the modern C# interface, including these recent changes.

C# has a unified type system. Every (non-interface) type inherits from the object type. This was a core part of C# 1.0, and it remains so even after two decades of language transformations.

Arrays are arguably the most important data structures beyond the primitive types in any programming languages. C#'s array is a generic type, which supports all common array operations such as indexing, in a type safe way. Arrays, and other collection types, now support range-based slicing syntax (`..`) as well, using the relatively new Index and Range .NET types.

Another important, and relatively new, pair of types in .NET are Spans and ReadOnlySpans, which are also tightly integrated into the C#

language. These types provide a safe way to access and manipulate the objects in memory, and their elements or members, without having to rely on expensive memory operations.

Designing and implementing a well-behaved type entails a lot of work. It sometimes requires writing a lot of boilerplate code just to create a simple type in many OOP-focused languages like Java and C#. The virtual methods defined in the `object` class are some of those examples. In principle, but not necessarily in practice, many of these methods need to be overridden, among many other methods, which makes creating a new type tedious, above everything else.

One of the biggest trends or themes in C#'s evolution has been to make this task of creating custom types easier. For example, `struct`, which has been a part of C# since version 1.0, is just a special kind of `class`, with some compiler-generated code, so that they do not need to be manually implemented by developers. The recent additions, records and record structs, are another example of such a trend in C#.

In the .NET world, tuples (or, value tuples) are the easiest compound types to create and use. These are really "lightweight" classes, whose use should be preferred over creating more formal types in many cases. An enum type also allows creating a simple, special kind of, type that is based on a set of constant values.

Despite all the recent changes, `class` is still the most important construct in C#. Programming in C# is fundamentally about defining types, e.g., using `class` (and, its cousins like `record` and `struct`). In the Classes chapter, we go through the essential elements of C# `class`. It should be noted that a lot of what we describe in the context of `class` also apply to `record` and `struct`.

Records are used to create immutable reference types, which are primarily for simple data storage. C# provides a concise syntax for creating and using records. In particular, it supports the positional parameters for creating record types. For example, creating a type is now as simple as just declaring a record constructor. Compiler automatically generates an entire class and a number of methods.

```
record Book(string Title, string Author);
```

Structs are value types, based on the base class `ValueType`, which in turn inherits from `Object`. Struct types themselves do not support inheritance. There are also special kinds of structs such as readonly structs, ref structs, and record structs.

C# supports extension methods, which are static methods associated with other existing types. Extension methods can be called as if they were instance methods of the target types.

In the first part of the book, we primarily focus on the types. In the rest of the book, we cover the rest of the important constructs in C#, such as expressions and statements, and exceptions.

The `new` operator is one of the most important components of C# expressions. They are used to create an array, or a new value or object of a specified type, e.g., `class` or `struct`. We also go through the object initializer and collection initializer syntax in this chapter.

There are many different kinds of expressions in C#, in addition to those commonly found in other programming language. Expressions, and operators, are discussed throughout the book, but some of the more common expressions are collected in the Expressions chapter. A lambda expression is an anonymous function. They are defined using the "fat arrow" `=>` operator. Lambda functions can be passed as an argument to other methods.

C#'s statements are more or less the same as those found in other C-style languages, with some minor differences. We quickly go through all statements in C# in the Statements chapter, mainly for completeness.

One of the notable changes with the `switch` statement, which was originally based on C's `switch - case` statement, is that it now supports pattern matching, along with `is` and `switch` expressions. Pattern matching was initially popularized by functional programming languages like Haskell, and it is now an integral part of many (functional or imperative) programming languages, including C#.

The `using` statements, and declarations, are used to manage resources that require cleanup after use, such as file handles or database connections. When an object which implements the `IDisposable` interface goes out of scope, its `Dispose` method is automatically called.

Finally, we conclude this Mini Reference with some quick descriptions of C# exceptions and attributes. Exceptions are thrown using the `throw` expression, and they indicate unusual or exceptional situations during the execution of a program. In C#, exception handling is implemented using the `try` statement. An attribute is a declarative tag that is used to convey information to runtime about the behavior of various elements in your code, such as classes, methods, and properties. C# attributes are just regular classes inherited from `System.Attribute`.

One thing to note is that, although we cover a broad range of topics in this book, we still leave out some important parts of C#, to keep the book to a manageable size in the spirit of the "mini reference".

These include LINQ (language-integrated query), expression trees, reflection, async programming, iterators and async stream, generic math, and delegates and .NET `Func` and `Action` types, among other things. We also leave out many commonly used .NET types such as Lists, Sets, and Dictionaries, which are really a core part of C#. The readers are encouraged to consult other resources for these topics and beyond. Furthermore, we do not discuss unsafe code in this book.

Chapter 2. C# Programs

A C# program can comprise one or more source code files (called the compilation units), e.g., in one or more directories on file system. All source files in a program (executable or library) are compiled together, and hence there are no particular orders among the source files.

Each source code file may contain a set of special declarations and custom type definitions. Type definitions may be included in zero or more nested namespaces. A `namespace` in C# is a logical organizational unit, which is primarily used to reduce the chance of name collisions.

2.1. Main Methods

An executable C# program, or an "application", should either include a type declaration (e.g., class or interface) that has a `Main()` static method or include a source file which includes the "top-level statements". This is the entry point to the program.

The `Main()` static method can return a value of type `int`, or it may not return any value (indicated by the `void` return type). They can accept an argument of type `string[]`, or they may be defined without any method parameters. The `Main()` methods may be synchronous, or they can be declared as `async` and return `Task` or `Task<int>`.

The valid signatures for the `Main()` static methods are, therefore, the following eight combinations:

```
static void Main() { /* ... */ }
static int Main() { /* ... */ }
static void Main(string[] args) { /* ... */ }
static int Main(string[] args) { /* ... */ }
static async Task Main() { /* ... */ }
static async Task<int> Main() { /* ... */ }
static async Task Main(string[] args) { /* ... */ }
static async Task<int> Main(string[] args) { /* ... */ }
```

2.2. Top-Level Statements

Since C# 9, one source file in a program may use top-level statements as long as the program does not include any other entry point methods. It is a syntactic sugar to help remove some boilerplate code. The top-level statements are automatically wrapped in the `Main` method of an implicitly defined startup class.

- Top-level statements can access the command line arguments using the implicitly defined variable `args` of type `string[]`.
- If the top-level statements return an integer value, that value becomes the integer return code to the operating system/runtime.
- The top-level statements may contain `async` expressions.

For example,

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

A file including this one line of code, using the implicit global using feature available since C# 10, is more or less equivalent to something like the following:

```
namespace Namespace1 {
    class Program1 {
        static void Main(string[] args) {
            Console.WriteLine("Hello World!");
        }
    }
}
```

As of C# 11, there are some restrictions:

- Only one file in your application may use top-level statements.
- All top-level statements in that file should be placed before any other namespace members such as type declarations.
- A program cannot have both top-level statements and an explicit entry point, e.g., a `Main` method.

2.3. Declarations

A C# program comprises a series of namespace declarations. A namespace can include other declarations, or "members". When the non-namespace member declarations are included in a program with no explicitly specified enclosing namespace, they belong to the "global namespace".

Namespace members can be type declarations or other (nested) namespace declarations. Type declarations are used to define classes, structs, records, enums, interfaces, and delegates. A type declaration can include other members. For instance, class declarations can contain declarations for constants, fields, properties, events, indexers, operators, methods, instance and static constructors, finalizers, and other nested types.

2.4. Access Control

Access of the top-level type declarations (e.g., within a namespace) can be controlled using one of the following accessibility declarations:

`public`	Access is not limited within a given program.
`internal`	Access is limited to this assembly. This is the default access level for the top-level types.

`file`	Accessible only from the types declared in this same source file. (New in C# 11.) When a type has the `file` modifier, it is said to be a file-local type.

Note that namespaces do not provide access limitations. That is, they are all implicitly `public`.

Access of the members of the type declarations (top-level or nested) is largely limited by the accessibility of their containing types, and it can be additionally controlled using some of the following access modifiers, depending on the types:

`public`	Access not limited.
`protected internal`	Accessible within this assembly *or* from the types derived from the containing class.
`internal`	Access limited within the current assembly.
`protected`	Access limited to the containing class or the types derived from the containing class.
`private protected`	Accessible from the containing class and its derived types which are *within this same assembly*.
`private`	Accessible from the containing type only.

Note that

- All members of an interface or an enum are public by default, and access modifiers cannot be used.
- Any access modifiers can be used for the members of a class.

- Only `public`, `internal`, and `private` can be used for the members of a struct.
- The default access level is `private` for the members of a class, record, and struct.

2.5. Blocks and Scopes

2.5.1. Scopes

C# is a lexically scoped language. The scope of a name is *statically* determined to be a particular region of program text, within which the entity declared by the name can be referred to without qualification of the name.

Scopes can be nested, and an inner scope may redeclare the meaning of a name from an outer scope. The name from the outer scope is then said to be "hidden" in the region of program text covered by the inner scope, and access to the hidden name is only possible by using the qualified name.

2.5.2. Blocks

A block statement permits multiple statements to be written in contexts where a single statement is allowed. A block consists of zero, one, or more statements written between the delimiters `{` and `}`. For example,

```
{
    STATEMENTS                     ①
}
```

① The `STATEMENTS` consists of zero, one, or more statements.

Note that the scope of a local variable or constant declared in a block is effectively the same block.

Chapter 3. Lexical Analysis

C# uses a 16-bit encoding of Unicode code points in character and string values. A Unicode escape sequence, `\U` or `\u` followed by a hexadecimal number, can be used to represent a Unicode code point in certain lexical contexts. That is, the Unicode escape sequences are processed in character literals, string literals, interpolated string expressions, and identifiers (but, not in keywords). For example, the sequence `\u0061` in a string literal `"\u0061pple"` represents a lowercase Alphabet `a`. Hence this string literal is equivalent to `"apple"`.

3.1. Lexical Elements

The following 5 elements make up the lexical structure of a C# program:

- Line terminators,
- White space,
- Comments,
- Compiler, or pre-processing, directives, and
- Tokens such as keywords and literals.

Compiler directives allow selective compilation of a program text, but otherwise they have no impact on its syntactic structure.

3.1.1. Line terminators

Line terminator elements divide the characters of a C# compilation unit into lines. Line terminators are:

- Carriage return (`\u000D`),
- Line feed (`\u000A`),
- Next line character (`\u0085`),

- Line separator (`\u2028`), and
- Paragraph separator (`\u2029`).

3.1.2. White space

White space in C# is defined as

- Any character with Unicode class Zs, including the ASCII space character (`\u0020`),
- The horizontal tab character (`\u0009`),
- The vertical tab character (`\u000B`), and
- The form feed character (`\u000C`).

3.1.3. Comments

C# supports two forms of comments, the C-style delimited comments and the C++-style single-line comments.

- A delimited comment begins and ends with the character sequences `/*` and `*/`, respectively.
- A single-line comment begins with the character sequence `//` and extends to the end of the same line.

```
/* Delimited comment can span
multiple lines */
using /* Or, even just a portion of a line */ System;

// A single line comment.
Console.WriteLine("Hello /* Not a comment */ World.");
```

Note that comments cannot be included in the string literals. White spaces and comments act as separators for tokens.

3.2. Tokens

Of the 5 basic lexical elements listed in the previous section, only tokens are significant in the syntactic grammar of a C# program. Tokens can be classified into identifiers, keywords, literals, operators, and other punctuation symbols. Literals are explained in the next section.

3.2.1. Identifiers

The C# identifiers comprise Unicode characters, and the rules of identifiers are generally similar to those found in other C-style languages. For example, an identifier can start with a letter or an underscore `_` and it can include other letters and numbers, etc. There are a few things to note:

- Unicode escape sequences are permitted in identifiers.
- Use of more than one consecutive underscores (e.g., `x__y`) is, although legal, discouraged.
- An identifier can be optionally preceded with a single `@` character.
- The `@`-form verbatim identifiers are typically used to "escape" the C# keywords, and otherwise its use is discouraged.

3.2.2. Keywords

Keywords are a set of reserved names in C# which cannot be used as identifiers in a program (except as `@`-prefixed verbatim identifiers).

```
abstract    as          base        bool        break
byte        case        catch       char        checked
class       const       continue    decimal     default
delegate    do          double      else        enum
event       explicit    extern      false       finally
fixed       float       for         foreach     goto
if          implicit    in          int         interface
```

```
internal   is         lock         long        namespace
new        null       object       operator    out
override   params     private      protected   public
readonly   ref        return       sbyte       sealed
short      sizeof     stackalloc   static      string
struct     switch     this         throw       true
try        typeof     uint         ulong       unchecked
unsafe     ushort     using        virtual     void
volatile   while
```

Contextual keywords are another set of names in C# which have special meanings only in certain contexts. These contextual keywords cannot be used as identifiers in those special contexts (again, except as `@` -prefixed verbatim identifiers).

```
add        alias      and          ascending   async
await      by         descending   dynamic     equals
file       from       get          global      group
into       join       let          nameof      nint
not        on         or           orderby     partial
remove     select     set          unint       value
var        when       where        yield
```

3.2.3. Operators and other punctuations

Operators are used to indicate the operations involving one or more operands in Punctuations are used for grouping and separating.

```
{    }    [    ]    (     )    .    ,    :    ;
+    -    *    /    %     &    |    ^    !    ~
=    <    >    ?    ??    ::   ++   --   &&
->   ==   !=   <=   >=    +=   -=   *=   /=   %=
&=   =    ^=   <<   <<=   =>   >>   >>=
```

3.3. Literals

Another important class of tokens is literals, which textually represent (constant) values of certain builtin types. For example, the `default` literal represents the default value of a type, which is discussed later. Interpolated strings are also lexically literals, but they are (non-constant) expressions in general.

3.3.1. The `null` literal

The `null` literal represents a null value. The `null` value itself does not have a type, but it can be converted to any nullable reference type or nullable value type through a null literal conversion.

3.3.2. Boolean literals

There are two Boolean literal values in C#, `true` and `false`. The type of these bool literals is `bool`.

3.3.3. Integer literals

Integer literals are used to write values of `int`, `uint`, `long`, and `ulong`. They have three possible forms: decimal, hexadecimal, and binary.

```
1_234                        ①
5678u                        ②
-10000000L                   ③
10_000_000uL                 ④
```

① An `int` integer literal.

② A `uint` integer literal. Uses the suffix, `u` or `U`.

③ A `long` integer literal. Uses the suffix, `l` or `L`.

④ A `ulong` integer literal. Uses a combination of `u/U` and `l/L`.

Note that (semantically insignificant) underscores _ can be used in the integer literals except for the beginning and ending positions. Underscores cannot be repeated one after another.

3.3.4. Real literals

Real literals are used to write values of floating point types `float` and `double` as well as `decimal`.

```
1_234.567                    ①
1000d                        ②
10.5e5f                      ③
19.99m                       ④
```

① A `double` literal.

② A `double` literal. Can use the suffix, `d` or `D`.

③ A `float` literal. Uses the suffix, `f` or `F`.

④ A `decimal` literal. Uses the suffix, `m` or `M`.

Non-consecutive underscores _ can be used in the real literals as well, except for the beginning and ending positions of both integer and fraction parts. Underscores do not affect the values of the real literals.

3.3.5. Character literals

A `char` literal corresponds to a single Unicode character. It is lexically represented as a character in quotes, as in `'a'` or `'B'`. A hexadecimal escape sequence represents a single Unicode UTF-16 code unit.

3.3.6. String literals

C# supports a few different forms of string literals. They are discussed in detail in the strings chapter.

3.4. Compiler Directives

In C#, the complier directives, also known as "pre-processing directives", are processed as part of the lexical analysis phase (e.g., and not as a separate pre-processing step). They can be used to conditionally skip sections of compilation units, to report error and warning conditions, or to mark certain regions of source code.

The following pre-processing directives are available:

`#define`, `#undef`

Used to define and undefine conditional compilation symbols.

`#if`, `#elif`, `#else`, `#endif`

Used to skip sections of code based on the compilation symbols.

`#line`

Used to control line numbers emitted for errors and warnings. Since C# 10, the `#line` directive can accept up to 5 decimal numbers as arguments, e.g., the start line, start character, end line, end character, and character offset.

`#error`

Used to issue errors or warnings.

`#nullable`

Used to enable or disable the nullable context.

`#region`, `#endregion`

Used to explicitly mark the beginning and end of a section of source code, respectively. Primarily to be used by IDEs.

`#pragma`

Used to specify predefined compilation symbols by the compiler.

Chapter 4. Namespaces

A compilation unit, e.g., a C# source code file, is organized according to the following linear structure:

- Zero or more `extern` alias directives,
- Zero or more global `using` directives (which affect all compilation units in the program),
- Zero or more (source file-specific) `using` directives,
- Zero or more global attributes, and
- Zero or more namespaces and their member declarations, or
- A file-scoped namespace and its member declarations.

C# programs use, by default, the (anonymous) global namespace unless the member declarations are included in an explicit namespace declaration (file-scoped or otherwise).

4.1. Namespace Declarations

As of C# 10 and later, a namespace can be declared with or without an explicit block. The scope of the namespace without a block extends to the entire compilation unit/file. The "normal" block-scoped namespace consists of the keyword `namespace`, a namespace name and body, and an optional semicolon `;`. For example,

```
namespace War {                          ①
  // Any member declarations
}
```

① The name of this namespace is `War`. Note that multiple (possibly nested) namespaces with explicit blocks can be included in a single compilation unit.

The file-scoped namespace, on the other hand, starts with the keyword `namespace`, a name, and a semicolon `;`. The rest of the compilation unit follows the same general structure, e.g., zero or more `extern` alias directives, `using` directives (but, not `global using` directives), followed by any member type declarations, and they all belong to this namespace. For instance,

```
namespace Peace;                    ①
// ...
```

① This namespace declaration extends to the end of the source file. If a source file includes a (possibly-nested) file-scoped namespace declaration, it cannot include any other namespace declarations in the same compilation unit.

Nested file-scoped namespaces can be declared with the dot qualified names. For example,

```
namespace Crime.Punishment;
// ...
```

The above declaration is equivalent to the following, for instance:

```
namespace Crime {
  namespace Punishment {
    // ...
  }
}
```

4.2. `extern` Alias Directives

An `extern` alias directive introduces an alias for an external namespace. For example,

```
extern alias Fruit;                        ①
Fruit::Apple apple;                        ②
```

① It introduces an external alias `Fruit`.

② `Apple` is a type declared within the namespace `Fruit`. The namespace/type names can be used in this compilation unit as if they are locally declared.

4.3. `using` Directives

Using the `using` directives, one can use namespaces and types defined in other namespaces without having to fully qualify them. The scope of a (non-global) `using` directive extends over the member declarations within the innermost enclosing namespace (including a file-scoped namespace), if any, or within the whole compilation unit otherwise.

4.3.1. The `using` namespace directive

A `using` namespace directive imports the types of a given namespace into the current namespace body (or, the compilation unit). The types contained in the using-declared namespace can be referenced directly.

```
using NAMESPACE_NAME ;
```

For example,

```
namespace Olympic.Soccer { class Match {} };
namespace Stats {
    using Olympic.Soccer;
    class SoccerStats {
        IList<Match> Matches { get; set; }
    }
}
```

4.3.2. The `using` static directive

A `using static` directive imports the nested types and static members of a type so that the names of the members and types can be used without qualification.

```
using static TYPE_NAME ;
```

For example,

```
namespace Arithmetic {
    public class Adder {
        public static void Add(int a, int b) => a + b;
    }
}
namespace App {
    using static Arithmetic.Adder;
    class Calculator {
        public int Sum(int a, int b) => Add(a, b);
    }
}
```

4.3.3. The `using` alias directive

A `using` alias directive introduces an identifier, or an alias, for a namespace or type within the innermost containing namespace (or, within the compilation unit). The identifiers introduced by the `using` alias directive can be used to reference the given namespace or type.

```
using IDENTIFIER = NAMESPACE_OR_TYPE_NAME ;
```

For instance,

```
namespace Europe.Ukraine {
    public class Citizen {}
}
namespace World {
    using Person = Europe.Ukraine.Citizen;
    class Farmer: Person {}
}
```

4.4. Global `using`

The `using` declarations can be made effective across an entire program, using the keyword `global`. The `global using` declarations can be used for any of the three `using` declaration types, `using` namespace, `using static`, and `using` alias. But, `global using` can only be used in a compilation unit, but not within an explicitly declared namespace, and its scope extends over the entire program. For example,

```
global using Machine;
global using static Robot.Arm;
global using Orange = Modern.Black;
```

4.5. Implicit `using`

Starting with C# 10, one can automatically add common `global using` directives for the C# project you are building, using the implicit usings feature. The "implicit" aspect refers to the fact that the `global using` directives are added to a generated file in the project's *obj* directory.

In case of the `Microsoft.NET.Sdk`, for example, the following namespaces are implicitly declared with `global using`:

- `System`
- `System.Collections.Generic`
- `System.IO`
- `System.Linq`
- `System.Net.Http`
- `System.Threading`
- `System.Threading.Tasks`

4.6. Member Declarations

As indicated before, as of C# 9 and later, top-level statements can be used in one of the source code files in a program in liu of a type that includes the `Main` method, e.g., as an entry point to the program. With this exception (which is merely a syntactic sugar), all statements in a namespace (explicitly declared or otherwise) must be either other (nested) namespace declarations or type declarations.

A type declaration is

- A `class` declaration,
- A `struct` declaration,
- A `record` declaration,
- An `enum` declaration,
- An `interface` declaration, or
- A `delegate` declaration.

Access to these namespace top-level type declarations can be explicitly controlled using the access modifiers. Otherwise, their default access level is `internal`.

Chapter 5. C# Type System

C#, and .NET, has a unified type system. Every type in the C# language inherits, either directly or indirectly, from the `object` base type (`System.Object` on .NET).

The types in C# are divided into two main categories, reference types and value types. Variables of the value types may directly contain their data, whereas variables of the reference types store references to their data. Values of reference types are treated as objects simply by viewing the values as type `object`. On the other hand, values of value types are treated as objects by performing *boxing* and *unboxing* operations.

Generic types are discussed later. Generic type parameters can designate either a value type or a reference type.

5.1. Value Types

C#'s value types can be divided into simple types, enum types, struct types (including record structs), and nullable types. Simple types comprise the builtin `bool` type and other numeric types. Nullable value types can contain a `null` value.

By default, when assigning a value, passing an argument to a method, or returning a method result, the values are copied. Note, however, that if a value type contains a data member of a reference type, the reference to the instance of the reference type is copied.

5.1.1. The `System.ValueType` type

A value type cannot explicitly derive from other types. All value types implicitly inherit from the class `System.ValueType`, which itself is not a value type. `System.ValueType`, in turn, inherits from the ultimate base class, `System.Object`.

5.1.2. Default constructors

All value types implicitly declare a public parameterless instance constructor called the default constructor. The default constructor returns a zero-initialized instance known as the type's default value

5.1.3. Boxing and Unboxing

Through boxing and unboxing, a value of any type, value type or reference type, can be treated as `object`. This provides a unified view of the C# type system. Boxing a value of a value type creates an object on the heap, which can be referenced via a reference variable. Unboxing does the reverse operation. That is, it returns the value contained in the object referenced by the given variable.

5.1.4. Nullable value types

A nullable value type `T?` can represent a `null` value in addition to all values of its underlying (non-nullable) value type `T`. This syntax `T?` is shorthand for the generic `System.Nullable<T>` struct. For example, you can assign any of the following three values to a `bool?` variable: `true`, `false`, or `null`. An instance of a nullable value type `T?` has two public readonly properties:

```
public bool HasValue { get; }
public T Value { get; }
```

5.2. Reference Types

All types in C#, including reference types, derive from the .NET base class `System.Object`. With value types, each variable has its own copy of the data. With reference types, on the other hand, two variables can reference the same object. Operations on one variable can therefore affect the object referenced by other variables.

C# provides three built-in reference types, `object`, `string`, and `dynamic`. Furthermore, custom reference types can be declared with the following keywords:

- `array`,
- `class`,
- `record` (or, `record class`),
- `interface`, and
- `delegate`.

Array types are somewhat special in that they do not have to be declared before they are used. Array types are constructed by appending square brackets to a type name. For example, `int[]` is a (single-dimensional) array of `int`, `int[,]` is a two-dimensional array of `int`, and `int[][]` is an array of arrays of `int`.

5.2.1. Nullable reference types

In the nullable context, all variables of a reference type are non-nullable. You can explicitly declare a nullable variable by appending the type name with the `?` symbol, which denotes a nullable reference type. For example,

```
string firstName = "Pele";             ①
string? middleName = null;             ②
```

① The variable `firstName` is non-nullable.

② `string?` is a nullable counterpart of the reference type `string`. The variable `middleName` is nullable.

For nullable reference types, the compiler uses flow analysis to ensure that any variable of a nullable reference type is checked against `null` before it's accessed or assigned to a non-nullable reference type.

5.3. Dynamic Types

All C# types, builtin or user-defined, use static binding. That is, the type of a variable or an expression is determined at compile time. The `dynamic` type is an exception. It uses dynamic binding at run time. In most cases, `dynamic` can be considered identical to `object` (since all types ultimately inherit from `object`).

For example,

```
dynamic theSecret = 42;
dynamic theGreatestUnknown = "meh";
```

Syntactically, an expression of the `dynamic` type can be implicitly converted to any type at run time. If the conversion fails, however, a runtime exception is thrown. If dynamic binding is not desired, the expression can be converted to `object` first, and then to the desired type. For instance,

```
dynamic realSecret = 42 * 42 * 42;
int huh = realSecret;                   ①
string oops = realSecret;               ②
```

① This implicit conversion will succeed at run time.

② This will throw an exception at run time.

```
dynamic knownUnknown = "moo...";
var ox = (string)(object) knownUnknown;   ①
var cow = (int)(object) knownUnknown;     ②
```

① Conversion succeeds at compile time.

② Build fails for this conversion.

Chapter 6. Constants

A constant is a variable representing a constant value, that is, a value that can be computed at compile time. Constant members are discussed later, e.g., in the class constant member section.

Local constants can be declared using a local constant declaration statement.

```
const TYPE NAME = EXPR ;                ①
```

① A constant is declared with the keyword `const`. The words with all upper case letters, in notations like this, are placeholders. That is, `TYPE`, `NAME`, and `EXPR` will need to be replaced with a type, name (or, identifier), and expression in a real statement. In the local constant declaration, the right hand side `EXPR` must be a constant expression.

More than one constant can be declared in a single statement. For example,

```
const TYPE NAME1 = EXPR1 , NAME2 = EXPR2 ;
```

The above statement is equivalent to the following two declarations:

```
const TYPE NAME1 = EXP1 ;
const TYPE NAME2 = EXP2 ;
```

The scope of a local constant is the block in which the declaration occurs, similar to local variables. Within the scope of a local constant, it is a compile-time error to declare another local variable or constant with the same name.

Chapter 7. Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable.

7.1. Variable Categories

Variables belong to a few different categories:

- Local variables,
- Static variables,
- Instance variables, and
- Array elements.

In addition, function parameters are also variables.

Local Variables

A local variable can be declared with a local variable declaration in a block. A local variable declared this way is not automatically initialized and thus has no initial value.

Static Variables

A field declared with the `static` modifier is a static variable. The initial value of a static variable is the default value of its type.

Instance Variables

A non-static field is an instance variable. The initial value of an instance variable of a class (or, record or struct) is the default value of the variable’s type.

Array Elements

The initial value of each of the elements of an array is the default value of the type of the array elements.

7.2. Local Variable Declarations

7.2.1. Local variables

A local variable declaration declares one or more local variables, similar to a local constant declaration:

```
TYPE NAME ;                                   ①
TYPE NAME1 , NAME2 ;                          ②
```

① As indicated, we use this somewhat unusual notation to represent C# syntax in this book. `TYPE` and `NAME` need to be replaced with a proper type and a valid variable name in a real statement, respectively. The statement ends with a semicolon `;`.

② More than one variable can be declared in one statement with the same type. This particular example only includes two variables, but it illustrates that the variables are separated by commas `,`.

Variables can be declared with initializers.

```
TYPE NAME = EXPR ;
TYPE NAME1 = EXPR1 , NAME2 = EXPR2 ;
```

When all variables have initializers, the explicit type name can be omitted in lieu of `var`. For example,

```
var NAME = EXPR ;
var NAME1 = EXPR1 , NAME2 = EXPR2 ;
```

The identifier `var` is a contextual keyword in the context of local variable declarations. The type of an implicitly typed local variable is inferred from the type of the associated initializer expression.

A discard variable `_` may be used as an implicitly declared variable, for example, when you intend to ignore the result of an expression. Discard variables are often used in deconstruction, or in pattern matching expressions. Here's a trivial example.

```
var _ = 1 + 2;
```

7.2.2. Ref local variables

A `ref` local variable is used to refer to values returned using `return ref`.

```
ref TYPE NAME = REF_EXPRESSION ;
ref var NAME = REF_EXPRESSION ;
```

You define a `ref local` by using the `ref` keyword in two places:

- Before the variable declaration.
- Immediately before the call to the method that returns the value by reference.

7.2.3. Ref readonly local variables

A `ref readonly` local variable is similar to a `ref` local variable, but they are not writeable.

```
ref readonly TYPE NAME = REF_EXPRESSION ;
ref readonly var NAME = REF_EXPRESSION ;
```

Note that you cannot declare "readonly local" variables in C# (which some people might consider a serious limitation considering that the C# language provides so much support for immutability *everywhere else*).

Chapter 8. Local Functions

Local constants and local variables can be declared in members of a type. Likewise, one can declare local functions and static local functions anywhere local constants and variables can be declared. The difference between static and non-static local functions is

- (Non-static) local functions can use, and capture, variables from the enclosing scope. That is, local functions are closures.
- Static local functions, on the other hand, cannot reference the outside variables.

lambda functions are essentially *anonymous* local functions. These two syntactic forms are more or less interchangeable, but using one or the other may be preferred depending on the context.

8.1. Local Functions

A local function is declared as follows:

- Local function header, comprising
    - Optional attributes, and optional local function modifiers,
    - The return type or `void`,
    - The function name,
    - A type parameter list, in case of a generic local function,
    - A formal parameter list in a pair of parentheses, followed by
    - An optional type parameter constraint clause, and
- Local function body, comprising
    - A function body block, or
    - An expression body ending with a semicolon `;`.

For example,

```
static (int, int) DoAdditionTwice(int a, int b) {
    var fstTime = LocalAdd();
    var sndTime = LocalAdd();
    return (fst, snd);

    void LocalAdd() => a + b;           ①
}
```

① The `LocalAdd` local function uses the local variables of the enclosing scope, namely, the two parameters of `DoAdditionTwice`.

Local functions are closures just like lambda functions. Local function modifiers can be either `async` or `static`, or both.

8.2. Static Local Functions

Static local functions are a special form of local functions. They are explicitly prohibited from using, or capturing, local variables, parameters, and `this` from the enclosing scope. They cannot use instance members of the type. Syntactically, a static local function declaration includes the `static` modifier. For example,

```
static int AbsoluteValue(int a) {
    return LocalAbs(a);

    static void LocalAbs(int x) =>
        x > 0 ? x : -x;                 ①
}
```

① The `LocalAbs` local function does not use any variables of the enclosing scope, and hence it can be declared as `static`.

Chapter 9. Formal Parameters

Function parameters are also variables. A parameter comes into existence when we invoke an anonymous function or a function member (e.g., method, constructor, accessor, or operator). Parameters may be initialized with values of the *arguments* given in the invocation. *Function arguments* are normally passed by value. But, they can use *pass-by-reference* semantics using the keywords, `in`, `out`, or `ref`.

9.1. Parameter List

The formal parameter list consists of comma-separated fixed parameters and an optional `params` array parameter at the end. A fixed parameter consists of

- Optional attributes,
- An optional modifier, `in`, `out`, `ref`, `this`, `in this`, or `ref this`,
- The parameter type,
- The parameter name, followed by
- An optional default argument (`= VALUE`).

Each fixed parameter declares a variable of the given type with the given name, which is local to the function body.

A fixed parameter with a default value is known as optional parameter. Arguments are required for non-optional parameters. Optional parameters, if any, can only be placed after any required parameters, before the optional `params` array. Parameters with `in`, `out`, `ref`, `this`, `in this`, or `ref this` modifiers cannot have default arguments.

The `this` modifiers, namely, `this`, `in this`, or `ref this`, are only allowed on the first parameter of a static method in a non-generic, non-nested static class. Such a method is called an extension method.

9.2. Argument List

Invocation of a function or method is an expression.

```
EXPRESSION ( ARGUMENT_LIST )
```

`EXPRESSION` should be "callable", e.g., a method or a value of a delegate type. `ARGUMENT_LIST` can be empty. It follows more or less the same structure as the parameter list, and each given argument must have a corresponding parameter in the parameter list.

C# supports two kinds of argument syntax, *positional arguments* and *named arguments*. A named argument has a form `PARAMETER_NAME : VALUE`, whereas only `VALUE` is specified for a positional argument. When an argument is in the same position in the argument list as the corresponding parameter in the parameter list, it can use either named or positional argument syntax. Otherwise, only the named argument syntax is allowed. For example, given the following function,

```
static void Metodo(int a = 1, char b = 'b') {
    Console.WriteLine($"a = {a}; b = {b}");
}
```

We can call this function in any of the following ways:

```
Metodo();                                       ①
```

```
Metodo(2);                                      ②
Metodo(a: 2);
Metodo(b: 'c');                         ③
```

```
Metodo(4, 'd');                         ④
Metodo(a: 4, 'd');                      ⑤
Metodo(4, b: 'd');
Metodo(a: 4, b: 'd');
Metodo(b: 'd', a: 4);                   ⑥
```

① Both parameters are optional, hence `a:1` and `b:'b'` in this call.

② Positional vs name argument syntax. These two method invocations are equivalent to each other, with `a:2` and `b:'b'`.

③ The position of this argument is `0` whereas the position of its corresponding parameter is `1`. Hence, this particular method call cannot use the positional syntax for the parameter `b`. This call is equivalent to `Metodo(1, b: 'c')` or `Metodo(a: 1, b: 'c')`.

④ All following five method calls are equivalent, with `a:4` and `b:'d'`.

⑤ The method argument `a` in this call is called the "non-trailing named argument" because it precedes a positional argument, `'d'` (for `b`).

⑥ Both `a` and `b` in this call must use the named argument syntax.

9.3. The `params` Parameter

The last parameter in a formal parameter list in C# can be a `params` array parameter, which consists of

- Optional attributes,
- The keyword `params`,
- A one-dimensional array type, and
- A parameter name.

The `params` parameter is a value parameter. (But, note that `array` is a reference type.) In a method invocation, an argument for the `params` parameter can be specified either as an array value or as zero, one, or more arguments of the array's element type. For example, given the following function,

```
static int AddAll(params int[] operands) => operands.Sum();
```

It can be called with an `int` array argument,

```
var ints = new int[] { 1, 2, 3 };
var sum1 = AddAll(ints);
Console.WriteLine($"sum = {sum1}");
```

Alternatively, the function can be called as follows:

```
var sum = AddAll(1, 2, 3);
Console.WriteLine($"sum = {sum}");
```

Note that these two forms of function invocations are completely equivalent, despite the apparent syntactic difference. That is, for example, calling this function with an empty argument list (since the `params` parameter is always optional) and calling it with an empty array will return the same result.

9.4. Value Parameters

A formal parameter declared without any parameter modifiers is a value parameter by default. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the function invocation. The argument therefore should be an expression that is implicitly convertible to the parameter type.

9.5. in Parameters

Parameters with the in, out, or ref modifiers are passed by reference. The in keyword specifies that the called method does not modify the value. An in parameter corresponds to a local ref readonly variable. The in parameter is declared using the in modifier in the parameter specification, and it can include a default value. For example,

```
static void PrintCoord(in Coord cord) => ①
    Console.WriteLine($"Lat = {cord.Lat}; Lon = {cord.Lon}");
record struct Coord(float Lat, float Lon);
```

① The in parameter cord uses the call-by-reference semantics. The PrintCoord function cannot modify its value.

This function can be called with or without the in keyword.

```
var coord = new Coord(40.0f, -100.0f);
PrintCoord(coord);
PrintCoord(in coord);
```

Although it uses reference semantics, it cannot change the value of the in argument. Therefore, one can expect certain performance increase without having to worry about data mutability.

9.6. out Parameters

A parameter declared with the out modifier is an output parameter. The corresponding argument is passed by reference. For example,

```
static void SetCoord(out Coord c) =>
    c = new(0.0f, 0.0f);              ①
```

① The out parameter cannot be referenced before it is assigned, and it must be assigned a value before the function returns.

To use an out parameter, both the method definition and the calling method must explicitly use the out keyword. You can declare a new out variable in the argument list of a method invocation. For instance,

```
SetCoord(out var c);                        ①
Console.WriteLine($"Lat = {c.Lat}; Lon = {c.Lon}");
```

① We can also use the explicit type, e.g., `SetCoord(out Coord c);`. This is more or less equivalent to `Coord c; SetCoord(out c);`.

9.7. ref Parameters

The ref parameter works in a way similar to the out parameter. But, unlike in the case of out parameters, an argument that is passed to a ref, or in, parameter must be initialized before it's passed in. To use a ref parameter, both the method definition and the method invocation must explicitly include the ref keyword. For example,

```
static void SwapCoords(ref Coord c1, ref Coord c2) =>
    (c1, c2) = (c2, c1);
```

Then, we can call this method as follows:

```
Coord coord1 = new(10.0f, 20.0f),
      coord2 = new(-30.0f, -60.0f);
SwapCoords(ref coord1, ref coord2);         ①
```

① This will swap the values of coord1 and coord2. Note that Coord is a value type (a struct), and this would not have been possible but for the ref semantics.

9.8. Returns by Reference

9.8.1. Ref returns

C# methods, by default, return values in the similar way that arguments are passed by value. In contrast, *ref returns* are values that a method returns by reference to the caller. That is, instead of copying the value returned by a method, its reference is returned. (Or, more precisely, its reference is copied.) There are three aspects for the ref returns to work.

- The return type in the method signature needs to be prefixed with `ref`. For example, `public ref MyStruct MyMethod()`.
- The expression of the return statement in the method body needs to by marked as `ref`. For example, `return ref updatedStruct`.
- In order for the caller to modify the object's state, the ref return value needs to be assigned to a `ref` local variable.

9.8.2. Ref readonly returns

The called method may also declare the return value as `ref readonly` to return the value by reference, and at the same time enforce that the calling code cannot modify the returned value. The calling method can avoid copying the returned value by storing the value in a local `ref readonly` variable. For example,

```
static class RefReturnDemo {
    static ref readonly Coord Move(ref Coord moving) =>
        ref moving;    // return ref
    public static void PrintAfterMove(Coord here) {
        ref readonly var t = ref Move(ref here);
        Console.WriteLine($"Lat = {t.Lat}; Lon = {t.Lon}");
    }
}
```

Chapter 10. Builtin Value Types

We will go through a few simple types in this chapter. The builtin `string` type is discussed in the next chapter.

10.1. The `bool` Type

The C# `bool` type is an alias for the .NET `System.Boolean` struct type, whose value can be either `true` or `false`. The default value of the `bool` type is `false`. That is, `default(bool)` returns `false`.

10.1.1. The nullable `bool?` type

The nullable `bool?` type can be used for the three-valued logic, for example, when you work with databases that support a three-valued Boolean type. The predefined `&` and `|` operators support the three-valued logic for the `bool?` operands.

10.2. The `char` Type

The C# `char` type is an alias for the .NET `System.Char` struct type, which represents a Unicode character in UTF-16 encoding. The default value of the `char` type is `\0`, that is, `\u0000`.

10.2.1. Char literals

You can specify a `char` value with:

- A character literal.
- A Unicode escape sequence, or
- A hexadecimal escape sequence, e.g., `\x` followed by the hexadecimal representation of a character code.

10.3. Integral Types

C# supports 11 integral types: Four unsigned integer types, `byte`, `ushort`, `uint`, and `ulong`, and four signed integer types, `sbyte`, `short`, `int`, and `long`, which are represented using two's complement format. The `char` type is also an integral type. In addition, C# includes two native integer types, `nint` and `nunit`.

`byte` (`System.Byte`)

Unsigned 8-bit integers with values from 0 to 255 (`== Pow(2,8)-1`).

`sbyte` (`System.SByte`)

Signed 8-bit integers with values from -128 (`== -Pow(2,(8-1))`) to 127 (`== Pow(2,(8-1))-1`).

`ushort` (`System.UInt16`)

Unsigned 16-bit integers with values from 0 to 65535.

`short` (`System.Int16`)

Signed 16-bit integers with values from -32768 to 32767.

`uint` (`System.UInt32`)

Unsigned 32-bit integers with values from 0 to 4294967295.

`int` (`System.Int32`)

Signed 32-bit integers with values from -2147483648 to 2147483647.

`ulong` (`System.UInt64`)

Unsigned 64-bit integers with values from 0 to 18446744073709551615.

`long` (`System.Int64`)

Signed 64-bit integers with values from -9223372036854775808 to 9223372036854775807.

The `checked` and `unchecked` operators and statements are used to control overflow checking for integral-type arithmetic operations and conversions.

10.4. Native Integer Types

C# now include two new new contextual keywords, `nint` and `nuint`, which represent native signed and native unsigned integer types, respectively, which are intended for use in low-level libraries. These contextual keywords are only treated as keywords when name lookup does not find a viable result at that program location.

The C# types `nint` and `nuint` correspond to the underlying .NET types `System.IntPtr` and `System.UIntPtr`, respectively, with compiler support for additional conversions and operations for those types as native integer types on a specific platform, or operating system.

10.5. Floating Point Types

C# supports two floating-point types, `float` and `double`, which are represented using the 32-bit single-precision and 64-bit double-precision IEC 60559 formats, respectively. Floating point number literals are described in the Lexical Analysis chapter.

10.6. The `decimal` Type

The `decimal` type is a 128-bit real number type, suitable for high-precision calculations. Internally a decimal is represented as an integer scaled by a power of ten. Decimal numbers are guaranteed to have at least 28-digit precision. The decimal type has greater precision but may have a smaller range than the floating-point types.

If a decimal arithmetic operation produces a result whose magnitude is too large for `decimal`, a `System.OverflowException` is thrown.

Chapter 11. Strings

The C# `string` keyword is an alias for the `System.String` type in .NET. A string represents text, which is implemented as a read-only sequence of `char` objects. The `Length` property of a string returns the number of `char` objects, and not the number of Unicode characters. Likewise, the `[]` operator can be used for read-only access to individual chars.

Strings are immutable. That is, you cannot update the value of a string object. The `+` operator concatenates strings and returns a new string. The equality operators `==` and `!=` are defined to compare the values of string objects. Hence, although `string` is a reference type, strings behave more like values rather than references. Empty strings can be initialized with the constant, `Empty`, or the empty string literal, `""`.

```
var s1 = String.Empty, s2 = "";
```

11.1. Quoted String Literals

String literals can be written in three different forms in C#, namely, quoted, verbatim, and raw string literals. In addition, there is a special literal syntax for "UTF-8 encoded strings", which is new in C# 11. Quoted string literals start and end with double quotation marks `""`. A quoted string literal must escape certain special characters, e.g., `\n` for a newline and `\t` for a tab, etc.

11.2. Verbatim String Literals

Verbatim string literals start with `@` and they are also enclosed in double quote characters. They are sometimes called "@-quoted strings". Special characters in verbatim string literals need not, and should not, be escaped, which make them easier to write. In particular, verbatim string literals preserve new line characters as part of the string text. Hence, verbatim strings are convenient for multi-line strings, or strings that contain many **backslash characters**, etc. A double quotation mark in an @-quoted string can be escaped as two quotation marks `""`.

For example,

```
Console.WriteLine(@"""Hello world"" is
     really a cliche now.");
```

This statement will print out

```
"Hello world" is
     really a cliche now.                 ①
```

① Notice that the leading white space characters as well as the newline at the end of the first line are included in the output.

11.3. Raw String Literals

A new form of raw string literal has been introduced in C# 11, which starts with a three or more double-quote character sequence, e.g., `"""`, and ends with the same character sequence, e.g., with the same number of double quotes.

The opening double-quote sequence can be immediately followed by an optional new line, in which case the enclosed string content should also end with a new line. These new lines are *not* part of the string literal. When the opening double-quote sequence is not followed by a new line, the raw string literal cannot include new lines, and the closing double quote sequence should be on the same line.

For a *multiline raw string literal*, the starting position of the ending double-quote sequence sets the "trim line" for the enclosed string literal content. That is, the spaces before that position in the string literal are not considered part of the string. For example,

```
var xml = """"
          <element attr="content">
            <body>My body</body>
          </element>
          """";
```

11.4. UTF8 String Literals

C# represents all strings using UTF16 encoding. Hence it requires encoding and decoding to produce and consume UTF8 encoded strings (e.g.,`byte[]`), which are commonly used, for example, for communication purposes. As of C# 11, one can now write an UTF8-encoded string in a literal form in a C# program. Syntactically, it is a regular string literal with a suffix `u8` or `U8`.

```
var str = "Heelloo, cliche~~"u8;
```

It should be noted that, despite the name, the type of a UTF8 string literal is `ReadOnlySpan<byte>`, not `string`, which is essentially a readonly span over a byte array representing a UTF8-encoded string.

11.5. String Interpolation

Interpolated strings are identified by the `$` prefix and they can include (to-be-)interpolated expressions in braces `{}`. String interpolation achieves the same results as the `String.Format` method, but it is generally preferred because it often improves the readability and maintainability of code.

11.5.1. Constant interpolated strings

Since C# 10, when all interpolated expressions in an interpolated string are constant, it is considered a constant expression.

```
const int year = 2022;
const string wc = $"World{" "}Cup {year}";
const string greeting = $"Welcome to {wc}!";
```

11.5.2. Verbatim string interpolation

C# also allows verbatim string interpolation, that is, string interpolation over verbatim string literals, using the `$@` or `@$` syntax. For instance,

```
var myAnswer = "Huh?";
var conversation = @$"Tom: What do you say?
        Me: {myAnswer} {myAnswer}";
```

11.5.3. Raw string interpolation

String interpolation can be used with (single-line or multi-line) raw string literals as well. In such a case, the number of dollar signs (`$`) present at the start of the literal determines the number of braces (`{`) needed to start an interpolation expression. Any brace sequence with fewer braces than that is just treated as content, and they are included in the output. For example,

```
var json = $$"""
             {
                "content": { "title": "Raw Oyster" },
                "length" : {{value.Length}}
             };
             """
```

Chapter 12. Generics

C#, and .NET, allows defining a set of related types with type parameters, commonly known as *generics*. A "constructed type" of a generic type is generated at compile time, if necessary, e.g., with particular type arguments. A constructed type can be used in most places in the language in which a type name can appear. The most common use of generics is to construct collection types.

In general, generics help maximize code reuse, type safety, and performance. The .NET class library, for example, contains several generic collection classes in the `System.Collections.Generic` namespace. In C#, one can declare generic interfaces, generic classes, generic records, generic structs, generic local functions, generic methods, generic events, and generic delegates, among others.

12.1. Generic Type Parameters

12.1.1. Type parameters

A generic type (and, a generic method, etc.) includes one or more type parameters (within angular brackets `<>`). By declaring particular types for these type parameters, a specific constructed type is created.

A type parameter is an identifier designating a value type or reference type that the parameter is to be bound to. Since a type parameter can be instantiated with many different type arguments, type parameters have slightly different operations and restrictions than other (real) types. For example, a type parameter cannot be used directly to declare a base class or interface in a generic type declaration.

As a type, type parameters are purely a compile-time construct. At run time, each type parameter is bound to a run-time type that was specified by supplying a type argument to the generic type declaration.

The run-time execution of all statements and expressions involving type parameters uses the type that was supplied as the type argument for that parameter.

12.1.2. Constraints on type parameters

Type constraints indicate what capabilities a type argument should have. Constraints are specified using a contextual keyword, `where`. If a type argument is used that does not satisfy the given constraint, the compiler will issue an error (or, a warning in some cases).

12.2. Type Parameter Constraints

The support for generic type constraints, both in terms of syntax and semantics, has been greatly extended over the past few C# releases. The following types of type parameter constraints are currently supported as of C# 11.

12.2.1. The `notnull` constraint

```
class C<T> where T : notnull { }
```

The type argument `T` must be a non-nullable type. You can use the notnull constraint to specify that the type argument must be a non-nullable value type or non-nullable reference type. Unlike most other constraints, if a type argument violates the notnull constraint, the compiler generates a warning instead of an error.

12.2.2. The `default` constraint

```
class C<T> where T : default { }
```

The `default` constraint implies that the base method does not have a class or struct constraint. This constraint resolves the ambiguity when you need to specify an unconstrained type parameter when you override a method or provide an explicit interface implementation.

12.2.3. The `new()` constraint

```
class C<T> where T : new() { }
```

The `new()` constraint asserts that `T` must have a public parameterless constructor. When used together with other constraints, the `new()` constraint must be specified last.

12.2.4. The `enum` constraint

```
class C<T> where T : System.Enum { }
```

You can use any `System.Enum` types as a base class constraint. With the enum constraint, the type parameter `T` can be used with the `System.Enum` static methods. `T` can also be used in a context where a specific enum type is expected.

12.2.5. The `delegate` constraints

```
class C1<T> where T : System.Delegate { }
class C2<T> where T : System.MulticastDelegate { }
```

The generic type argument `T` must be a `System.Delegate` or `System.MulticastDelegate` type. The delegate constraint enables you to write code that works with delegates in a type-safe manner.

12.2.6. The `struct` constraint

```
class C<T> where T : struct { }
```

The type argument `T` must be a non-nullable value type. Because all value types have an accessible parameterless constructor, the `struct` constraint implies the `new()` constraint, and they cannot be combined.

12.2.7. The `class` constraint

```
class C<T> where T : class { }
```

This constraint asserts that `T` must be a non-nullable reference type. It applies to `class`, `record`, `interface`, `delegate`, or `array`. When a type argument is a nullable reference type, a warning is issued.

12.2.8. The `class?` constraint

```
class C<T> where T : class? { }
```

It asserts that `T` must be a reference type, either nullable or non-nullable.

12.2.9. Interface type constraints

```
class C<T> where T : I1, I2, I3<T> { }
```

Specific interfaces, one or more, can be specified as part of a type constraint. In such a case, the type argument must be either an interface that inherits, or a non-nullable type that implements, the given interfaces, e.g., `I1`, `I2`, and `I3<T>` in this example.

As shown, the constraining interfaces can be generic. `T`, in `I3<T>`, must be a `non-nullable type` that implements the specified interface, `I3`.

```
class C<T> where T : I1?, I2?, I3<T>? { }
```

This constraint is equivalent to the previous example, `T : I1, I2, I3<T>`, except that `T` can be a nullable or non-nullable reference type as well as a non-nullable value type.

12.2.10. Base class type constraints

```
class C<T> where T : B1 { }
```

The type argument must be, or derive, from the specified base class, e.g., `B1`. `T` must be a non-nullable reference type derived from `B1`.

```
class C<T> where T : B1? { }
```

Likewise, the type argument must be, or derive, from the specified base class, `B1`. `T` may be a nullable or non-nullable type.

12.2.11. Specific type constraints

```
class C<T, U> where T : U { }
```

The type argument `T` must be, or derive from, the type `U`.

- If `U` is a non-nullable reference type, `T` must be non-nullable reference type.
- If `U` is a nullable reference type, `T` may be nullable or non-nullable.

12.3. Type Variance in Generics

In C#, type variance enables implicit reference conversion for array types, delegate types, and generic type parameters so that broader, or more specific, types can be used. Variances, either covariance or contravariance, are supported for reference types, but they are not supported for value types.

12.3.1. Covariance

Covariance enables you to use a more derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement covariant interfaces and implicit conversion of delegate types. An interface that has a covariant type parameter enables its methods to return more derived types than those specified by the type parameter.

As an example, because type `T` is covariant in `IEnumerable<T>` and `string` inherits from `object`, an object of the `IEnumerable<string>` type, for instance, can be implicitly assigned to an object of the `IEnumerable<object>` type.

12.3.2. Contravariance

Contravariance enables you to use a less derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement contravariant interfaces and implicit conversion of delegate types.

A type can be declared contravariant in a generic interface or delegate only if it defines the type of a method's parameters and not of a method's return type. `In`, `ref`, and `out` parameters must be invariant, meaning they are neither covariant nor contravariant.

Chapter 13. Interfaces

An `interface` essentially defines a behavioral contract, which can be implemented by classes, records, and structs, in term of its members:

- Methods,
- Properties,
- Indexers, and
- Events.

An interface does not generally provide implementations of the members it defines. Those members must be explicitly implemented by classes, records, or structs that implement the interface. A single (non-interface) type may implement multiple interfaces.

For the last few releases of C#, the concept of `interface` has been gradually generalized. An `interface` can now include

- Constants,
- Operators,
- Static constructors,
- Static methods, properties, indexers, and events, in addition to the instance methods, properties, indexers, and events,
- Static fields, and
- Nested types.

Some of these members can have default implementations. Beginning with C# 11, the concept of the "interface contract" has been further broadened. The contract is no longer limited to an object but rather it is extended over the relationship between two or more objects. In particular, C# now supports static abstract or virtual methods in interfaces, with or without default implementations.

13.1. Interface Declarations

An interface declaration statement declares a new interface type:

- A header including
  - Optional attributes, and other optional interface modifiers,
  - The keyword `interface` or `partial interface`,
  - The name of the interface,
  - A list of type parameters, if the interface is generic,
  - Any base interfaces, and
  - Type parameter constraints, if needed, followed by
- An interface body, which consists of
  - A list of member declarations written between the delimiters `{}`.
- An optional semicolon `;`.

For example,

```
public interface INode { void Append(INode node); }
```

An interface declaration which includes a type parameter list, or which is nested inside a generic class, record, or struct declaration, is a generic interface declaration. A generic interface declaration can include a type parameter constraint clause, if needed. For example,

```
interface Div<T> : Node<T> where T : INumber<T> {
    // ....
}
```

13.1.1. Interface modifiers

An interface declaration can optionally include an access modifier. In addition, the `new` modifier can be used within a class to indicate the interface hides an inherited member by the same name.

The `partial` type declarations are mainly used by code generators, and they are used to split a single type declaration over two or more source code files. An interface can also be declared as `unsafe`. We do not discuss unsafe code, as well as partial interface and partial class declarations and partial methods, in this book.

13.1.2. Type parameter variance

In case of interface and delegate declarations, type parameters can include variance annotations, namely, `in` and `out`.

- If the variance annotation is `out`, the type parameter is *covariant*,
- If the variance annotation is `in`, then it is *contravariant*, and
- Otherwise, the type parameter is *invariant* by default.

For example,

```
interface I1<out T1, in T2, T3> {}        ①
```

① `T1` is covariant, `T2` is contravariant, and `T3` is invariant.

13.1.3. Base interfaces

An interface can optionally inherit from other interface types. When at least one explicit base interface is specified, they are preceded by a colon `:`. The base interfaces of an interface comprises the explicitly inherited base interfaces and their base interfaces, defined recursively. An interface inherits all members of its base interfaces.

13.2. Interface Members

The members of an interface are those included in the interface declaration and the members of all base interfaces of the interface. An interface declaration declares zero or more members:

```
constants,
static constructors, static fields,
static or instance methods, static or instance properties,
static or instance indexers, static or instance events,
operators, and nested types.
```

All interface members implicitly have `public` access, but it can be changed using explicit access modifiers. An interface member can also include other modifiers such as `abstract` or `new`, depending on their specific types.

When an interface declares a method, property, indexer, or event member with the same name or signature as an inherited member, the derived interface member is said to *hide* the base interface member. To suppress a compiler warning, the declaration of the derived interface member can include a `new` modifier to indicate that the derived member is intended to hide the base member.

One can provide a default implementation for a method, property, indexer, event, and operator member of an interface. If an instance member has a default implementation, it is considered `virtual`. Otherwise, all instance members are `abstract` by default. In contrast, all static methods are `sealed` by default. But, they can be explicitly declared as `abstract` or `virtual`.

A static `Main` method of an interface can be used as an entry point to a program. All interface members can be decorated with attributes.

13.2.1. Interface method declarations

Interface methods are declared as follows:

- Optional attributes, and optional (access) modifiers,
- The return type, including `void`,
- The name of the method,
- A type parameter list, in case of a generic method,
- A formal parameter list, enclosed in parentheses, `(` and `)`,
- Type parameter constraints, if any, and
- A semicolon `;`, or a default implementation.

13.2.2. Interface property declarations

An interface property declaration consists of:

- Optional attributes, and optional (access) modifiers,
- The type of the property,
- The name of the property, followed by
- Interface accessors, enclosed in curly braces `{` and `}`.

The accessors of an interface property declaration can end with semicolons `;`, or they can have default implementations. An interface property accessor can be indicated as read-only, write-only, or read-write, using the keywords `get` and `set`. For example,

```
public interface IInterface {
    string ReadOnly { get; }
    string WriteOnly { set; }
    string ReadWrite { get; set; }
}
```

13.2.3. Interface indexer declarations

An interface indexer declaration:

- Optional attributes, and optional (access) modifiers,
- The type of the element,
- The keyword `this`,
- A formal parameter list, enclosed in *square brackets*, `[` and `]`, and
- Interface accessors, as defined above.

13.2.4. Interface event declarations

Interface events are declared as follows:

- Optional attributes, and optional (access) modifiers,
- The keyword `event`,
- The type and name of the event, and
- A semicolon `;`, or a default implementation.

13.2.5. Other member types

In addition, an interface can include constants, static constructors, static fields, operators, and other nested types. They are more or less identical, both syntactically and semantically, to those found in classes, records, and structs, with the exception of their default access modifiers. Static methods and operators are further discussed in the next section.

13.2.6. Interface member access

A method, property, or event member `M` of an interface `I` is accessed via member access expression of the form `I.M`. In case of an indexer argument list `A`, it is accessed as `I[A]`.

13.3. Interface Static Members

Static method, property, indexer, and event members, as well as operators, of an interface are implicitly non-virtual and `sealed`.

13.3.1. Static `abstract` and `virtual` members

As of C# 11, the static method, property, indexer, event, and operator members of an interface can now be declared as `abstract` or `virtual`. They can also be explicitly declared as `sealed` to indicate its non-abstract/non-virtual nature.

Classes, records, and structs that implement an interface with `abstract` members are required to provide implementations of these members. The static members, just like instance members, can then be accessed off of type parameters that are constrained by the interface.

13.3.2. Accessing static `abstract` interface members

When `T` is constrained by an interface `I` and `M` is an accessible static `abstract` or `virtual` member of `I`, `M` may be accessed on a type parameter `T` as `T.M`. For instance,

```
interface I1<T> {
    static virtual T Meth() => default;  ①
}
```

① A static virtual method, with a (literally) default implementation.

```
interface I2<T> {
    static abstract T Prop { get; }      ①
}
```

① A static abstract read-only property.

```
class C<T> : I2<T> where T : I1<T> {     ①
    public static T Prop => T.Meth();     ②
}
```

① `T : I1<T>` is an interface type constraint.

② Note the syntax, `T.Meth()`. We can use the type parameter `T` to access a static virtual or abstract member as if it is a real type.

13.3.3. An example and an informal explanation

This new feature will likely see many different uses in the coming days, but it will be instructive to review the primary use case that prompted this change in the first place. As far as .NET is concerned, the canonical example is generic math, which was released with C# 11 as a showcase for the use of static abstract/virtual members in an interface.

Let's do something similar for demonstration. Let's suppose that we have a number of different types for representing natural numbers. (And, in fact, we do in C#, e.g., `ushort`, `uint`, and `ulong`.) We may even want to add an infinite-precision natural number type in the future. Now, the task is to implement a `Sum` method, which takes a pair of arguments of a natural number type and returns its sum.

Just to be clear, it's not about math. We will end up using operator overloading (for `+`) in this example, but again it's not about operators or binary operations. It's really about an abstraction that does not naturally fit into the OOP style. If you want to design a negate function for a number type `A`, for instance, it is natural to declare it as an instance method for `A`. That is, we can do something like `a.negate()` for `a` of type `A`. But, what about addition? Like `a` plus `b`? In the hard-core OOP style, we will have to implement it as a method, for example, something like `a.Add(b)`. This is, however, rather unnatural since you have to arbitrarily pick one of the operands as the "object". It will be a lot more natural to implement it as a non-object oriented function, like

Add(a, b). In C#, static methods are really functions. (And, operators are just (special kind of) static methods.) Now, as of C# 11, we can do a lot more with static methods, including declaring them as part of public interface and being able to override them when necessary, etc.

Going back to our toy problem, let's start from the top. Here's the Sum function that we would like to ultimately implement.

```
public static class Demo<T> where T : INat<T> {
    public static T Sum(T l, T r) => l + r;
}
```

This static class Demo is implemented as a generic type with parameter T. The implementation of Sum relies on the fact that we can add two values of type T. We will need to add that requirement as a generic type constraint. In this example, we use an interface type constraint, T : INat<T>. So, how should we define INat<T>? Here's our solution:

```
public interface INat<T> where T : INat<T> {
    static abstract T operator +(T l, T r);
}
```

First of all, it should be noted that we could not have done this before C# 11 when static methods were not part of interface-defined API. This is only possible now because we have static virtual and static abstract methods that are a genuine part of the interface.

Second, there is a strange-looking pattern in this generic interface declaration. The type argument T of INat<T> is constrained to be a type that inherits or implements INat<T> itself. This is idiomatic. If you look at our Demo<T> above, T should be constrained to be of INat<T> to support the desired add (+) operation. Clearly, INat<T> itself has to support the same operation (as an interface static method) *with the same constraint* for T. Hence, this idiomatic pattern.

Let's create a type that implements `INat<T>`. Here's an example:

```
public struct NatNum : INat<NatNum> {
    public NatNum(uint v = 0u) => V = v;
    public uint V { get; set; }
    static NatNum INat<NatNum>.operator +(NatNum l, NatNum r)
=> new NatNum(l.V + r.V);
}
```

Now, the kicker is that the `Sum` function that we implemented earlier just works with this new type.

```
var (a, b) = (new NatNum(1), new NatNum(2));
var sum = Demo<NatNum>.Sum(a, b);
Console.WriteLine($"sum = {sum.V}");
```

We will leave it as an exercise to the readers to try and create a few different types that implement `INat<T>`, including an infinite precision natural number type, and test their values with the `Sum` function.

13.4. Default Implementations

An interface can provide default implementations for its (static or instance) members. A class, record, or struct that implements such an interface is required to have a single most specific implementation for the interface method, either directly implemented by the class, record, or struct, or inherited from its base types or interfaces.

This enables API authors to add methods to an interface in later versions without breaking source or binary compatibility with existing implementations of that interface.

13.5. Interface Implementations

C# also supports explicit interface member implementations, which use the fully qualified names for the interface members. For example,

```
public interface ICoder {
    void Code(string lang);
}
public class ChatBot: ICoder {
    void ICoder.Code(string lang) {
        // ...
    }
}
```

Explicit interface members can only be accessed via the interface type. For example, the implementation of `ICoder.Code` provided by the `ChatBot` class can only be invoked via the `ICoder` interface type.

```
ICoder smartChatBot = new ChatBot();     ①
smartChatBot.Code("C#");                 ②
```

① The variable, `smartChatBot`, is declared as the `ICoder` type, and not as `ChatBot`, in this example.

② We can call the method `Code` declared in the `ICoder` interface on this variable `smartChatBot`.

Chapter 14. Objects

`System.Object` is the ultimate base class of all .NET classes (including records, structs, and enums, etc.).

```
public class Object {}
```

Every method defined in `Object` is available in all objects. Derived classes can, and many do, override some of these methods.

For example,

`Equals`	Supports comparisons between two objects. The default implementation uses simple reference comparison.
`GetHashCode`	Generates a unique number corresponding to the value of the object to support the use of a hash table.
`ToString`	Returns a human-readable text string that describes an instance of the class.
`Finalize`	Performs cleanup operations before an object is garbage-collected. As stated in the beginning, the use of a `Dispose` method is generally preferred over finalizers.
`MemberwiseClone`	Creates a shallow copy of the current object.

In addition, `Object` includes the default and copy constructors as well as some other default implementations, which can be used by any class in the .NET system.

Chapter 15. Arrays

An `array` is a builtin reference type which contains zero, one, or more elements of a single type. An object of an array type can be dynamically created at run time using the `new` operator, e.g., by specifying the *length* of the new array and its element type. The length of an array object cannot be changed once created.

15.1. Array Types

An array type can be declared, syntactically, with an element type followed by a pair of square brackets (`[]`). E.g., `int[]`.

15.1.1. `System.Array`

All array types inherit from the abstract base class, `System.Array`, which itself is not an array type. An implicit reference conversion exists from any array type to `Array`. Likewise, an explicit reference conversion exists from `Array` to any array type.

15.1.2. Generic collection interfaces

A single-dimensional array `T[]` implements the interfaces `IList<T>` and `IReadOnlyList<T>`, and their base interfaces. Therefore, `T[]` can be implicitly converted to `IList<T>` or `IReadOnlyList<T>`, or to any of their base interfaces.

More generally, if a type `S` can be implicitly converted to a type `T`, then `S[]` implements both `IList<T>` and `IReadOnlyList<T>`, and `S[]` can be implicitly converted to `IList<T>` or `IReadOnlyList<T>`, or to any of their base interfaces. Likewise, if a type `S` can be explicitly converted to a type `T`, then can be explicitly converted to `IList<T>` or `IReadOnlyList<T>`, or to any of their base interfaces.

15.1.3. Multidimensional arrays

An array is an inherently *linear data structure*. Its elements are stored in a linear contiguous memory space. C# also supports multi-dimensional arrays as a generalization of one-dimensional arrays.

Syntactically, a series of commas are used between the square brackets of an array type. The number of commas plus one denotes the number of dimensions of the array type, called the *rank*. The following example allocates a one-dimensional, a two-dimensional, and a three-dimensional array.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

The `a1` array contains 10 elements, the `a2` array contains 50 (`10 * 5`) elements, and the `a3` array contains 100 (`10 * 5 * 2`) elements.

The element type of an array can be any type, including an array type. A one-dimensional array with elements of another one-dimensional array type is sometimes called a *jagged array*. The following example allocates an array of arrays of `int`:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

The first line creates an array with three elements, each of type `int[]` and each with an initial value of `null`. The subsequent lines then initialize the three elements with references to individual array instances of varying lengths.

15.2. Array Creation

The `new` operator initializes all elements of an array to the element type’s default value, which, for example, is "zero" for a numeric type and `null` for a nullable reference type. For example,

```
int[] a = new int[3];            ①
```

① All 3 elements of `a` are initialized with int `0`.

Alternatively, the `new` operator can be used with an array initializer, in which the initial values of the array elements are specified between a pair of curly braces `{` and `}`. The following example allocates and initializes an `int[]` with 3 elements.

```
int[] a = new int[] {1, 2, 3};   ①
```

① Since the length of the array can be inferred from the number of value between `{` and `}`, we need not explicitly specify the length.

Local variable and field declarations can be shortened further such that the array type does not have to be explicitly specified again.

```
int[] a = new[] {1, 2, 3};         ①
char[] b = {'x', 'y', 'z'};        ②
```

① No explicit type needs to be specified when the type is known.

② We can even drop `new[]`. This is a special syntax for arrays.

An empty array can be created as follows:

```
var empty = Array.Empty<int>();
```

15.3. Array Elements

15.3.1. Indexing

The elements of an array can be accessed through indices, from `0` to `length - 1`. For example,

```
int[] a1 = new[] { 1, 1, 2, 3, 5 };
for (var i = 0; i < a1.Length; i++) {
    Console.WriteLine($"{i}: {a1[i]}");  ①
}
```

① The loop variable `i` is used to access the `i`-th element of `a`.

An array can also be enumerated using a `foreach` statement.

15.3.2. Slicing

An array, and other sequence collection types, support both indexing and slicing operations, e.g., through the `System.Index` and `System.Range` types and their methods. For example,

```
int[] a2 = { 2, 4, 6, 8, 10 };
Array.ForEach(a2[..2], Console.WriteLine);      ①
Array.ForEach(a2[1..4], Console.WriteLine);     ②
Array.ForEach(a2[^2..], Console.WriteLine);     ③
```

① `a2[..2]` selects the first two items of `a2`.

② `a2[1..4]` selects items of indices, 1, 2, and 3.

③ `a2[^2..]` selects the last two items.

Chapter 16. Spans and ReadOnlySpans

System.Span<T> and System.ReadOnlySpan<T> enable read-write and read-only access to contiguous regions of memory, respectively. They are comparable to, and have similar performance characteristics of, the builtin array types.

16.1. System.Span<T>

Span<T> is a generic ref struct, a value type, representing a contiguous region of arbitrary memory in a type-safe and memory-safe way.

The Span<T> indexer is declared with a ref T return type, which provides semantics like that of indexing into arrays, returning a reference to the actual storage location rather than returning a copy of what lives at that location:

```
public ref T this[int index] { get { ... } }
```

16.2. System.ReadOnlySpan<T>

ReadOnlySpan<T> is also a generic ref struct similar to Span<T>, but it is readonly.

The ReadOnlySpan<T> indexer returns a ref readonly T instead of a ref T, enabling it to work with immutable data types like string. ReadOnlySpan<T> makes it very efficient to slice strings without allocating or copying.

```
public ref readonly T this[int index] { get { ... } }
```

Chapter 17. Tuples

C#'s tuples are a builtin value type corresponding to the generic types, `System.ValueTuple<T1, T2, …>`, on .NET. (Note that, in contrast, `System.Tuple<T1, T2, …>` are reference types.) A tuple provide concise syntax to group one or more data elements in a lightweight data structure. One of the most common use cases of a tuple is to "return multiple values" from a method (e.g., instead of using `out` parameters).

To define a tuple type, you specify types of all its fields and, optionally, the field names, within a pair of parentheses. Tuple elements are public fields. Hence tuples are *mutable* value types. For example,

```
var copa = (2022, "World Cup");         ①
(int age, bool isTall) = (21, true);    ②
```

① The variable `copa` is initialized with `(2022, "World Cup")`, whose type is `(int, string)` (or, `System.ValueTuple<int, string>`).

② A tuple `(21, true)`, whose type is `(int, bool)` (or, `System.ValueTuple<int, bool>`), is deconstructed and the variables `age` and `isTall` are initialized with the tuple's elements.

17.1. Tuple Fields

Unnamed tuple fields can be accessed with the default names, `Item1`, `Item2`, and so on., from the left. Any alternative, and presumably more meaningful, field names can be explicitly specified in a tuple initialization expression, using the `name:value` syntax. The field names can also be specified in the definition of a tuple type.

In some cases, the field names can be inferred from the names of the corresponding variables in a tuple initialization expression. This is known as tuple projection initializers. For example,

```
var t1 = ("dog", 9.5);                      ①
var t2 = (First: 3, Second: "world");       ②
int wins = 5;
int losses = 1;
var t3 = (wins, losses);                    ③
var x = new {Country: "USA", Code: "US"}
var t4 = (x.Country, x.Code);               ④
```

① The fields of `t1`, of type `(string, double)`, have the default names, `Item1` and `Item2`, respectively.

② The two fields of `t2` can be accessed as `t2.First` (or, `t2.Item1`) and `t2.Second` (or, `t2.Item2`), respectively. The field names are not part of the type.

③ The tuple field names can be inferred in expressions like this. The first field of `t3` has names `wins` or `Item1`. Likewise, the second field has names `losses` or `Item2`.

④ Likewise, the names of the fields of `t4` are `Country` or `Item1` and `Code` or `Item2`, respectively.

17.2. Tuple Equality

Tuple types support the `==` and `!=` operators. These operators compare their respective members following the order of tuple elements. Tuple assignment and tuple equality comparisons don't take field names into account.

Two tuples are comparable when both of the following conditions are satisfied:

- Both tuples have the same number of elements, and
- For each tuple position, the corresponding elements from the left-hand and right-hand tuple operands are comparable with the `==` and `!=` operators.

For example, all three `Console.WriteLine()` calls in the following example will print `true`:

```
Console.WriteLine((1, 2) == (1, 2));
(byte a, int b) t1 = (1, 2);
(short a, long b) t2 = (1, 2);
Console.WriteLine(t1 == t2);
var t3 = (Left: 1, Right: 2);
var t4 = (Right: 1, Left: 2);
Console.WriteLine(t3 == t4);
```

17.3. Tuple Deconstruction

A value of one tuple type can be assigned to another tuple type if:

- Both tuple types have the same number of elements, and
- Each element on the right hand side is assignable to the corresponding element on the left hand side.

C# supports deconstructing tuples, along with some other data types (e.g., records), which lets you unpackage all items in a tuple in a single operation. Here's a simple example of tuple assignment and deconstruction.

```
(short, short) point1 = (1, -1);                ①
(int, int) point2 = point1;                     ②
(uint, int) point3 = ((uint, int)) point1;      ③
long x1 = 0, y1 = 0;
(x1, y1) = point2;                              ④
```

① The type of variable `point1` is `(short, short)`.

② The value `point1` can be assigned to `point2` because `short` and `short` are implicitly convertible to `int` and `int`, respectively.

③ An explicit conversion is required to assign `point1` to `point3` because `short` is not implicitly convertible to `uint`.

④ A tuple can be deconstructed in assignment. The type of `x1` and `y1` is `long`. `int` can implicitly convertible to `long`.

As tuples are often used to "return multiple values" from a method, tuple deconstruction is also commonly used to assign the multiple returned values to multiple variables in one go.

17.3.1. Assignment and declaration in the same deconstruction

The assignment and initialization statements syntactically distinct, but conceptually they are more or less equivalent. Initializations involve variable declarations and initial value assignments. As of C# 10, one can combine assignment and initialization through deconstruction in one statement. For example,

```
int min = 0;                           ①
(min, int max) = (33, 77);             ②
```

① The variable `min` is declared here, and it is initialized with value `0`.

② This statement is a combination of an assignment (for the existing variable `min`) and a variable initialization (for a new variable `max`). This kind of statement is often informally called a "multiple assignment", regardless of whether variables are assigned or initialized, since multiple variables (`min` and `max` in this example) end up being assigned new values in one statement.

17.3.2. Discard variables

When deconstructing a tuple or calling a method with `out` parameters, you may not be interested in all elements of a given tuple. In this kind of scenarios, you can use *discards* to ignore certain variables that you do not intend to use. A discard is a *write-only* variable whose name is `_` (the underscore character). The discard variables can be used more than once in a single deconstruction or `out` variable.

For example,

```
static (int year, bool olympic, bool worldcup)
    SportsYear(int year) => (year, false, true);     ①
var (year, _, copa) = SportsYear(2026);               ②
Console.WriteLine($"Year: {year}, World Cup: {copa}");
```

① This function returns a tuple of type `(int, bool, bool)`.

② Since we are only interested in `year` and `worldcup`, we ignore the second element using a discard variable `_`.

17.3.3. `Deconstruct` method

C# has built-in support for deconstructing tuples, records, and DictionaryEntry types. One can also provide a similar deconstruction for any type by implementing a `Deconstruct` method. The method returns `void`, and each value to be deconstructed is indicated by an `out` parameter in the method signature.

The `Deconstruct` method can be overloaded to allow an object of the given type to be deconstructed into multiple different combinations of variables and types. One can also add a `Deconstruct` method to an existing type as an extension method.

Chapter 18. Enums

An `enum`, or enumeration, is a value type defined by a set of (named) constants. All `enum` types inherit from the `System.Enum` abstract base class, which in turn inherits from type `System.ValueType`. `Enum` is itself not an enum type, and automatic boxing and unboxing conversions are supported from any enum type to `Enum` and from `Enum` to an enum type, respectively.

18.1. Enum Declarations

To define a new `enum` type, an `enum` declaration is used, as follows:

- Optional attributes, and an optional access modifier,
- The `enum` keyword,
- An enum type name, and
- Optional underlying type specification, followed by
- An enum body that includes the named members of the enum.

The following example declares an enum type `Crypto` with three constant values, `Bitcoin`, `Ether`, and `Ripple`.

```
enum Crypto {
    Bitcoin,
    Ether,
    Ripple,                                    ①
}
```

① The trailing comma, e.g., for the last member, is optional.

These 3 constants have values of 3 consecutive numbers, `0`, `1`, and `2`, of the `int` type starting from `Bitcoin`. One can also assign an explicit integral value(s) to a specific member(s) using an assignment syntax.

Each member can be accessed through the usual member access syntax (the "dot notation"). For instance, we can refer to the zero-th element of `Crypto` as `Crypto.Bitcoin`.

18.2. Underlying Types

Each enum type has a corresponding *underlying type*. By default, an enum type’s underlying type is `int` unless it is explicitly specified in the enum declaration. The underlying type must be one of the *integral types* other than `char`. For instance, the following example declares an enum type named `USCoin` with an underlying type of `byte`.

```
enum USCoin: byte {
    Penny = 1, Nickel = 5, Dime = 10, Quarter = 25,
}
```

Unlike in many other programming languages supporting enumeration types, C#'s enums are rather permissive in terms of typing. In fact, the set of values that an enum type can take on is that of the underlying type, and it is not limited to the explicitly declared named members. For example, any value of the underlying type of an enum can be cast to the enum type, and it is a valid enum value of that type.

The reverse is also true. Any member of an enum type can also be explicitly cast to its underlying integral type. (Note, however, that an enum type is a distinct and separate type from its underlying type.) For instance, using the above `USCoin` example,

```
var nick = (short)USCoin.Nickel;          ①
var coin = (USCoin)10;                    ②
var doce = (USCoin)12;                    ③
```

① The value of `nick` is `5` of type `short`.

② The value of `coin` is `USCoin.Dime`.

③ The type of `doce` is `USCoin` and it has a value `12`.

The default value of any enum type is the integral value zero (`0`) converted to the enum type. The integer literal `0` is implicitly converted to any enum type regardless of whether a named constant corresponding to zero exists or not. For example, the following is a valid statement, e.g., without requiring an explicit casting:

```
USCoin none = 0;
```

18.3. Enum Modifiers

Enum types do not permit derivation. The modifiers `static`, `sealed`, and `abstract` therefore do not make sense and they are not allowed for `enum`. Otherwise, all other type and access modifiers can be used.

For example, the top-level `enum` declarations (either within a file scope or in a namespace) can be `public`, `file`, or `internal` (default). For nested enums, `new`, `public`, `protected`, `internal`, and `private` can be used. Note that enum members do not have any separately declared accessibility. All members of an enum type are accessible if that enum type is accessible.

18.4. Enum Members

An enum type declaration includes zero, one, or more enum members, whose names should be unique across the given enum type. Enum members are named and scoped in the same way as the fields within a class. The scope of an enum member is the body of its containing enum type. Within that scope, enum members can be referred to by their simple names.

Each enum member is associated with a *constant expression initializer*, whose type should be compatible with the underlying type of the containing enum type. The type of each member is the given underlying type, and its value should be in the valid range of this underlying type.

If an enum member is declared with no initializer, then its associated value is set implicitly:

- If it is the first member declared in the enum type, its associated value is `0`,
- Otherwise, the associated value of the enum member is obtained by increasing the associated value of the textually preceding enum member by `1`.

Multiple enum members can share the same associated value. For example,

```
enum Color : uint {
    Red = 5,                    ①
    Blue,                       ②
    Navy = Blue,                ③
    // Green = -10,             ④
}
```

① The value of the enum member `Red` is explicitly set to `5`.

② `Blue` is automatically assigned a constant value `6` since its immediate preceding member has value `5`.

③ The members `Color.Blue` and `Color.Navy` have the same `6`.

④ `Green` would have been an invalid member since its associated value is outside the range of `uint`.

All members of a given `enum` type can be iterated over, for instance, using the `foreach` statement. Here's a simple example using the above `Color` enum:

```
foreach(var c in Enum.GetValues(typeof(Color))) {
   Console.WriteLine($"Name: {Enum.GetName(typeof(Color), c)},
Value: {(uint)c}");
}
```

18.5. Enum Operations

An enum type declaration cannot include method or property members. But since an enum type (automatically) derives from the class `System.Enum`, it can use the inherited methods and properties of this class.

In particular, the following operators can be used on the values of enum types:

- Comparison operators `==`, `!=`, `<`, `>`, `<=`, `>=`,
- Binary `+`,
- Binary `-`,
- Logical `^`, `&`, `|`,
- Bitwise unary `~`,
- Prefix and postfix `++`, `--`, and
- The `sizeof` operator.

18.5.1. Extension methods on enums

Furthermore, extension methods can be defined over enum types.

Chapter 19. Classes

Classes are one of the most fundamental constructs in C# for creating a new (reference) type, which structurally defines the state (e.g., fields of a class) and possible behavior (e.g., methods of a class) of an instance of the type, also known as an *object*. A class may contain data members (constants and fields), function members (methods, properties, events, indexers, operators, instance and static constructors, and finalizers), and other nested types.

Class types support *inheritance* and *polymorphism*, a mechanism whereby *derived classes* can extend and specialize *base classes*. All members of a base class, except for constructors and finalizers, are inherited by derived types.

19.1. Class Declarations

A new class declaration comprises

- A header including
  - Optional attributes, and other optional class modifiers,
  - The keyword `class` or `partial class`,
  - The name of the class,
  - A list of type parameters, if the class is generic,
  - An explicit base class specification, if any,
  - Any interfaces that the class implements, and
  - Type parameter constraints, if needed, followed by
- A class body, which consists of
  - Member declarations written between the delimiters `{}`.
- A class declaration can end with an optional semicolon `;`.

The following is a declaration of a class named `Point`:

```
public class Point {
    private int x, y;                        ①
    public Point(int x, int y) {             ②
        this.x = x;
        this.y = y;
    }
}
```

① Private fields.

② A public instance constructor.

19.1.1. Class modifiers

A class declaration can include zero, one, or more access modifiers and/or other class modifiers: `new`, `abstract`, `sealed`, and `static`. The `new` modifier can be used for nested types only. It specifies that the class hides an inherited member by the same name.

19.1.2. Abstract classes

An `abstract` class cannot be instantiated (e.g., using the `new` operator), and it is only used as a base class for other classes. Only abstract classes can include abstract members. When a non-abstract class is derived from an abstract class, directly or indirectly, the derived class needs to provide the implementations of all inherited abstract members.

19.1.3. Sealed classes

A `sealed` class cannot be used as a base class for inheritance. Hence, it cannot include `abstract` or `virtual` members. All C# classes are non-sealed by default, but it is generally a good practice to mark the classes as `sealed` that are not specifically designed to be derived. The compiler can make certain optimizations on the `sealed` classes.

19.1.4. Static classes

The `static` modifier is used to declare a *static class*. A static class does not define a type, and it cannot be instantiated.

- A static class cannot be declared as `sealed` or `abstract`.
- A static class declaration cannot include the base class specification.
- A static class cannot cannot implement interfaces.
- A static class can contain only static members. (Note that constants and nested types are considered static members.)
- A static class cannot have `protected`, `protected internal`, or `private protected` members.

19.2. Type Parameters

A class declaration that includes a type parameter list is a generic class declaration. A type parameter is a placeholder for a type argument supplied to construct a new class. A type argument is substituted for each type parameter when constructing a specific type. Furthermore, any class nested inside a generic class, record, or struct declaration is itself a generic class declaration.

A class definition may specify a set of type parameters by following the class name with angular brackets `<>` enclosing a list of type parameter names. The type parameters can then be used in the body of the class declarations to define the members of the class. In the following example, `TFst` and `TSnd` are type parameters of a generic class `Pair`:

```
public class Pair<TFst,TSnd> {
    public TFst First;
    public TSnd Second;
}
```

As stated, when a generic class declaration is used to construct a type, a type argument must be provided for each type parameter:

```
var pair = new Pair<string, int> {        ①
    Fst = "number",
    Snd = 666,
};
var key = pair.Fst;                       ②
var val = pair.Snd;                       ③
```

① The variable `pair` is of a type `Pair<string, int>`. A generic type with specific type arguments such as `Pair<string, int>` is called a constructed type.

② `key` is a variable of the type `string` since `TFst` is `string`. `pair.Fst`, with a dot `.` operator, is a member access expression.

③ `val` is an `int` type since `TSnd` is `int`.

19.3. Base Classes

19.3.1. Inheritance

A class declaration may include a direct base class of a class type. Not specifying an explicit base class in the declaration is the same as deriving from type `object`. Except for class `object`, every class has exactly one direct base class. The base classes of a class are the direct base class and its base classes.

The direct base class of a class type shall be at least as accessible as the class type itself. For example,

```
public class A {}                         ①
class B : A {}                            ②
```

① The base class of A is implicitly object (System.Object).

② An internal class B derives from A, which has the same or broader accessibility than B (e.g, public vs internal). Class A is said to be the direct base class of B in this example.

```
class C<U,V> {}                      ①
class D<T> : C<int,T[]> {}           ②
```

① The base class of a generic class C<U,V> is object.

② The generic class D<T>, with type parameter T, inherits from a constructed class C<int,T[]>, which is a specialization of C<U,V> with U and V replaced with int and T[], respectively. The direct base class of a constructed class D<string>, for example, is a constructed, and non-generic, class C<int, string[]>.

A class inherits the members of its base class. Inheritance means that a class implicitly contains all members of its base class, except for the instance and static constructors, and the finalizer of the base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.

An implicit conversion exists from a class type to any of its base class types. Therefore, a variable of a class type can reference an instance of that class or an instance of any derived class.

19.3.2. Interface implementations

A class declaration can also (directly) specify one or more interface types that the class *implements*. In case of generic class declarations, each implemented interface type is obtained by substituting each type parameter in the given interface with the corresponding type argument of the constructed type. If there are more than one base class and/or interfaces specified in a class declaration, they are separated by commas ,.

For instance, the following example illustrates how a class can implement and extend constructed types:

```
class C<U, V> {}
interface I1<V> {}
class D : C<string, int>, I1<string> {}
class E<T> : C<int, T>, I1<T> {}
```

19.4. Static Constructors

C# supports both instance and static constructors. An *instance constructor* implements the actions required to initialize an instance of a class. A *static constructor*, on the other hand, implements the actions required to initialize the class itself.

A static constructor is declared like a method with no return type and the same name as the containing class. A static constructor declaration includes a `static` modifier. Static constructors are called by the runtime (e.g., .NET), and they cannot be called directly. For example,

```
public class A {
    public static A() {}                ①
    public static void Main() {}        ②
}
```

① The static constructor.

② If the class include a `Main` static method, the static constructor, if any, is called first before the `Main` method is executed.

- A static constructor declaration may include attributes and/or an `extern` modifier.
- An external static constructor declaration provides no actual implementation, and it ends with a semicolon `;`.

- For all other static constructor declarations, the body consists of a block `{}`.

19.5. Instance Constructors

Instance constructors are members of a class that prescribe any initialization actions of instances of the class. Instances are created using the `new` expression with an instance constructor, which allocates memory for a new instance, invokes the constructor, and returns a reference to the newly created instance.

The following statements create two `Point` objects and store references to those objects in two variables:

```
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```

The memory occupied by an object is automatically reclaimed when the object is no longer in use. It is neither necessary nor possible to explicitly deallocate objects in C#.

- An instance constructor declaration may include a set of attributes, access modifiers, and/or an `extern` modifier.
- An external constructor declaration provides no implementation, and its constructor body consists of a semicolon `;`.
- Otherwise, the constructor body is a block `{}`, which specifies the statements to initialize a new instance of the class.
- An instance constructor declaration may include an optional constructor initializer which specifies another instance constructor to invoke before executing the constructor body.

Instance constructors can be overloaded. For example, the `List<T>` class declares two instance constructors, one with no parameters and one that takes an `int` parameter. The following statements allocate two `List<string>` instances using each of the two constructors.

```
var list1 = new List<string>();
var list2 = new List<string>(10);
```

Unlike other members, constructors are not inherited. If no instance constructor is supplied for a class, then a default constructor, i.e., with an empty list of parameters, is automatically provided.

Alternative to the `new` operator expression, an object initializer creates an object of a given type and initializes fields or properties of the object in one statement.

19.6. Finalizers

A *finalizer* is a method member that implements the actions required to clean up an instance of a class, after it is used. Finalizers are not inherited. A class can have at most one finalizer. The finalizer for an instance is invoked automatically during garbage collection, and the timing of finalizer invocations is not deterministic. In general, the `using` statement provides a better approach to resource cleanup, and finalizers are not much used in the modern C# programming.

A finalizer is syntactically similar to a parameterless constructor, and it has the same name as type with tilde `~` as a prefix. Finalizers cannot have access modifiers. For example,

```
class WorkingClass {
    ~WorkingClass() {
        Console.WriteLine("Done working. For good.");
    }
}
```

19.7. Class Members

A class declaration may include constants, fields, methods, properties, events, indexers, operators, instance constructors, finalizers, static constructors, and nested types. The members of a class are

- The members declared in the class, and
- The members inherited from the base class.

The members inherited from the base class include

- The constants, fields, methods, properties, events, indexers, operators, and the nested types of the base class, but not
- The static and instance constructors and finalizers of the base class.

19.8. Constants

A constant is a class member that represents a constant value, that is, a value that can be computed at compile time. For example,

```
class A {
    public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

This declaration is equivalent to

```
class A {
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
}
```

19.9. Class Fields

A static or instance field is a variable that is associated with a class or with an instance of a class, respectively. For example,

```
class A {
    public static int A = 1, B = 10;      ①
    public string greeting = "hello";     ②
}
```

① A static field declaration.

② An instance field declaration. Note that an interface cannot include any instance fields.

This is equivalent to

```
class A {
    public static int A = 1;
    public static int B = 10;
    public string greeting = "hello";
}
```

Read-only fields may be declared with a `readonly` modifier. Assignment to a `readonly` field can only occur as part of the field's declaration or in a constructor in the same class.

19.10. Methods

19.10.1. Method member declarations

A method member implements a computation or action that can be performed by a class or an object of the class. A method is declared as follows:

- Method header, comprising
    - Optional attributes, and optional method modifiers,
    - Optional keyword `partial`,
    - The method return type or `void`,
    - The method name,
    - A type parameter list, if the method is generic,
    - A formal parameter list within parentheses `()`, followed by
    - An optional type parameter constraint clause, and
- Method body, comprising
    - A method body block,
    - An expression body ending with a semicolon `;`, or
    - A semicolon `;`.

As with many other declarations in C#, only certain combinations of these parts are syntactically valid. Allowed method modifiers are `new`, `static`, `virtual`, `sealed`, `override`, `abstract`, `extern`, and `async`, in addition to the access modifiers. Again, only valid combinations of these modifiers are allowed. A method with the `async` modifier is an `async` method.

19.10.2. Method parameters

Method parameters are described in an earlier chapter, Formal Parameters. Extension methods are discussed later.

19.10.3. Static and instance methods

When a method declaration includes the `static` modifier, that method is a static method. When no `static` modifier is present, the method is an instance method. An instance method operates on a given instance of a class, and that instance can be accessed as `this`.

19.10.4. Virtual methods

When an instance method declaration includes the `virtual` modifier, that method is a virtual method. The implementation of a virtual method can be superseded by derived classes. The process of superseding the implementation of an inherited virtual method is known as *overriding* that method.

19.10.5. Override methods

When an instance method declaration includes the `override` modifier, the method is said to be an `override` method. An override method overrides an inherited virtual method with the same signature. Only an `override` method can override an inherited method. Otherwise, the `new` modifier should be used, which makes the method *hide* the inherited method. If a method with the same signature of an inherited method is defined without the `new` modifier, the compiler will issue a warning.

19.10.6. Sealed methods

A sealed method, with `sealed override` modifier, overrides an inherited virtual method with the same signature. Sealed methods cannot be further overridden by its derived classes.

19.10.7. Abstract methods

A declaration of an instance method including the `abstract` modifier is an abstract method. An `abstract` method is implicitly a `virtual` method. Abstract methods are allowed only in abstract classes.

An abstract method declaration introduces a new virtual method but does not provide an implementation of that method, and its method body simply consists of a semicolon `;`. Non-abstract derived classes are required to provide their own implementation by overriding the abstract method.

19.10.8. External methods

The `extern` modifier declares external methods, which are implemented externally. The method body of an external method simply consists of a semicolon `;`.

19.10.9. Method body

The method body of a method declaration can be a block body, an expression body, or just a semicolon `;`.

```
class Oracle {
    public void Welcome() { }          ①
    public int Secret() => 42;         ②
    extern string Farewell(int secret);  ③
}
```

① The `Welcome` method has a block body, which is a block statement.

② The `Secret` method has an expression body, which consists of a fat arrow `=>` followed by an expression, an int literal `42` in this case.

③ The extern `Farewell` method has an empty body `;`.

If a method does not return a value, or if it returns an object of the `System.Threading.Tasks.Task` type in case of an `async` method, then the *effective return type* of the method is `void`. If a method returns a value of type `T`, or `Task<T>` in case of an `async` method, then the effective return type of the method is `T`.

- When the effective return type of a method is `void`,
    - If the method has a block body,
        - The `return` statements cannot include an expression.

        - If execution of the method completes normally, that method simply returns to its caller.
    - If the method has an expression body, `=> E`,
        - The expression `E` must be a statement expression, and the body is equivalent to a block body of the form `{ E; }`.
- When the effective return type of a method is not `void`,
    - If the method has a block body,
        - Each `return` statement must include an expression which is implicitly convertible to the effective return type.
        - Control cannot flow off the end of the method body.
    - If the method has an expression body, `=> E`,
        - The expression `E` must be implicitly convertible to the effective return type, and the body is equivalent to a block body of the form `{ return E; }`.

19.11. Properties

Properties are similar to fields, but they do not denote data storage locations. Rather, they are (direct or indirect) *accessor methods* to stored or computed data. Syntactically, properties are more like methods with some resemblance to fields.

- Property header, comprising
    - Optional attributes, and optional property modifiers,
    - The property type,
    - The property name, and
- Property body, comprising
    - An accessor block with an optional property initializer, or
    - An expression body, ending with a semicolon `;`.

Valid property modifiers are the same as those of methods, with the same or similar meanings. For example, properties can be static properties or instance properties, depending on whether they are declared with the `static` modifier or not. Likewise, properties can be `virtual` or `abstract`. Or, they can be declared as `new`, `override`, or `sealed override`, and so forth. Properties, and fields, can also be marked as `required`.

Property accessors can be one of `get`, `set`, `get; set`, and `get; init`. The corresponding properties are known as the "read-only", "write-only", "read-write", and "init-only" properties, respectively. Property initializers may only be specified for auto-implemented properties.

19.11.1. Expression-bodied properties

A property body may consist of an expression body, using the fat arrow `=>` syntax. For example,

```
class WitchOfTheWest {
    public int Age => 42;               ①
}
```

① The syntax is rather similar to that of a parameterless expression-bodied method, except for the absence of parentheses.

This expression body syntax declares a readonly property. That is, this class declaration is equivalent to the following:

```
class WitchOfTheWest {
    public int Age {
        get => 42;
    }
}
```

19.11.2. Auto properties

An automatically implemented property (or, auto-property for short), is a property with semicolon-only accessor bodies. An auto-property can be readonly (`get`), read-write (`get; set`), or init-only (`get; init`). When an auto-property is declared, the compiler automatically creates a hidden backing field, and implements the declared accessors using that backing field.

```
class WizardPuppy {
    public string? Name { get; }          ①
    public string? Breed { get; init; }   ②
    public string? Spell { get; set; }    ③
}
```

① An instance readonly auto-property.

② An instance init-only auto-property.

③ An instance read-write auto-property.

Note that, in this simplified example, all three properties can be `null`, or not explicitly set, when we first try to access any of these properties. Hence, they are declared as nullable types (e.g., nullable string).

An auto-property may optionally have a property initializer, which is essentially a variable initializer for the underlying backing field. For example,

```
class WizardPuppy {
    public string Name { get; } = "Toto";
    public string Breed { get; init; } = "Pug";
    public string Spell { get; set; } = "Meow";
    public WizardPuppy(string name) => Name = name;
}
```

Property values can be set, or updated, in the constructor as well, regardless of whether they are already explicitly set through property initializers or not. The read-write properties like `Spell` in this example can be updated, depending on their accessibility, in other methods and/or using the property accessor syntax.

Note that the readonly and init-only properties are both read-only properties. The only difference is that the values of init-only properties can be additionally set in object initializers whereas that is not permitted for readonly properties.

19.11.3. Accessor body

In general, the property body can be an accessor body, which consist of one or two accessor declarations:

- Accessor header, comprising
  - Optional attributes, and optional accessor access modifiers,
  - An accessor, e.g.,`get`, `set`, or `init`, and
- Accessor body, comprising
  - An accessor body block,
  - An expression body, ending with a semicolon `;`, or
  - A semicolon `;`, in case of `extern`, `abstract`, or auto-properties.

Valid sets of accessor member access modifiers are determined by the access modifier of the containing properties. The detailed rules are omitted, but in general, an accessor cannot be more accessible than its containing property.

```
public class WizardOfOz {
    private string residence = "Emerald City";
    private string vehicle = "Hot air balloon";
    public string Residence {                  ①
        get => residence;                      ②
        set => residence = value;              ③
    }

    internal string Vehicle {                  ④
        get {                                  ⑤
            return vehicle;                    ⑥
        }
        private set {                          ⑦
            vehicle = value;                   ⑧
        }
    }
}
```

① A public property of the `string` type, with a block body. Since it includes both `get` and `set` accessors, it is a read-write property.

② An expression-bodied get accessor. Its accessibility is `public` since its containing property `Residence` is public.

③ A `public` expression-bodied set accessor. Note that assignment is an expression in C#. The identifier `value` is a contextual keyword, representing the target value in an assignment expression. For example, in `wizard.Residence = "Omaha"`, the `value` is the string `"Omaha"`.

④ An `internal` read-write property.

⑤ A get accessor with a block body. Its accessibility is `internal`.

⑥ Although we simply return the value of a class instance field, `vehicle`, in this example, we can implement more complicated logic in this getter block body.

⑦ A `private` setter with a block body. The type of `value` is `string`, the same as that of its containing property, `Vehicle`. Note that one of the accessors can have a separate access modifier.

19.12. Indexers

An indexer is an instance member of a class that enables an object of the class to be indexed with a subscript `[]` operator, just like array objects. An indexer is declared like a property except that the name of the member is `this` followed by a parameter list between `[]`.

- Indexer header, comprising
  - Optional attributes, and optional indexer modifiers,
- Indexer declarator, comprising
  - The indexer element type,
  - The `this` or `INTERFACE.this`, followed by
  - A formal parameter list within square brackets `[]`, and
- Indexer body, comprising
  - An accessor definition block body, or
  - An expression body, ending with a semicolon `;`.

An indexer can also be read-only, write-only, or read-write. Valid indexer modifiers are the same as those of methods or properties, except that indexers cannot be declared as `static`. Indexers can be `virtual` or `abstract`, or `new`, `override`, `abstract override`, or `sealed override`, and so forth. The formal parameter list specifies the parameters of the indexer. Indexers can be overloaded just like method members. That is, a class can declare multiple indexers as long as their formal parameter lists are different.

19.13. Events

An event member enables an object or class to provide notifications. Event handlers can be attached to, and removed from, the event using the `+=` and `-=` operators, respectively.

An event is declared like a (static or instance) field except that the declaration includes the `event` keyword and that the type must be a delegate type. The event field stores a reference to a delegate that represents the event handlers that have been added to the event. If no event handlers are present, the field is `null`. The notion of raising an event is equivalent to invoking the delegate represented by the event.

Here's a simple end-to-end example:

```
delegate void PanicHandler(string message);     ①
```

① A delegate type to be used with our event example.

```
sealed class PanicButton {
    public event PanicHandler Click;            ①
    public PanicButton() =>                     ②
        Click += new PanicHandler(HandlePanic);
    public void DoPanic(string message) =>      ③
        Click(message);
    static void HandlePanic(string message) => ④
        Console.WriteLine(message);
}
```

① An `event` member of the delegate type `PanicHandler`.

② A public constructor, in which we initialize the event member `Click`. The `+=` operator is used to assign a delegate object, the `HandlePanic` static method in this example.

③ A test method used to raise an event. A delegate is callable.

④ An example event handler.

```
var button = new PanicButton();
button.DoPanic("We're not in Kansas anymore"); ①
```

① Testing the event. This will simply print out the message argument to the console, per our simple implementation.

19.14. Operators

An operator member is a `public` and `static` method that defines the meaning of applying a particular expression operator to instances of a class. Note that C# has a set of operators predeclared for overloading, and operator overloading does not use the new static method overriding framework.

19.15. Operator Overloading

There are three categories of overloadable operators.

Unary Operators

- The method takes a single formal parameter.
- The name comprises the keyword `operator` followed by an operator symbol:
  - `+`, `-`, `!`, `~`, `++`, `--`, `true`, and `false`.

Binary Operators

- The method takes two formal parameters.
- The name comprises `operator` and an operator symbol:
  - `+ , - , * , / , % , &, |, ^ , <<, >>, ==, !=, >, <, >=, <=`` ` ``

Conversion Operators

- The method takes a single formal parameter.
- The name comprises
  - Either `implicit operator` or `explicit operator` for implicit and explicit conversions, respectively, followed by
  - The class name.

19.16. Nested Types

A nested type is a type defined within another type. The default accessibility of a nested type is always `private` regardless of the kind of the containing type, e.g., class, record, struct, or interface.

For example,

```
public class Outer {
    struct Secret { }                    ①
    public class Shared { }              ②
}
```

① The accessibility of `Secret` is `private`, and it can only be accessed within the containing class `Outer`.

② `Shared` is declared as `public`. The full name of this nested type is `Outer.Shared`. For instance, an instance of this class can be created with `new Outer.Shared()`.

Nested types can be nested. For instance,

```
var voice = new Child.Inner.Voice(Tone: "Baritone");
public class Child {
    public class Inner {
        public record Voice(string Tone);
    }
}
```

A nested type can access all the members that are accessible to its containing type, including the containing type's `private` and `protected` members.

Chapter 20. Records

A `record class`, or `record` for short, is an immutable reference type that provides synthesized methods to support value semantics for equality. Records are primarily used for creating, storing, and manipulating simple, immutable data that contains a fixed set of values. C# provides a concise syntax for creating and using records. In particular, it supports the positional parameters for creating record types. That is, one can use the *positional record* syntax, to declare and create a record with init-only properties.

Records are distinct from classes in various respects. Most importantly, records supports *value-based equality* by default. Two objects of a record type are equal if the record type definitions are identical, and if for every field, the values in both records are equal. On the other hand, two variables of a class type, which uses the *reference-based equality* by default, are equal if the objects they refer to are the same object.

When you define a record type, the compiler synthesizes several members of the record, e.g., for copying and comparing records:

- Methods for value-based equality comparisons,
- Override for `System.Object.GetHashCode`,
- Copy and Clone members, and
- `PrintMembers` and `System.Object.ToString`.

Furthermore, when you declare a positional record, the compiler synthesizes additional methods:

- A primary constructor whose parameters match the positional parameters on the record declaration,
- Public init-only properties for the positional parameters, and
- A `Deconstruct` method to extract properties from the record.

Here's an example of a record class using the positional syntax:

```
public record Book(string Title, decimal Price);
```

This record can be used as follows, for instance:

```
var book = new Book("C# Mini Reference", 9.99m);
```

20.1. Record Declaration

A new record class can be declared in two different ways: With and without a record body. Here's a record declaration syntax.

- A header including
  - Optional attributes, and optional record modifiers,
  - The keyword `record`, `record class`, `partial record`, or `partial record class`,
  - The name of the record,
  - An optional type parameter list,
  - An optional positional parameter list,
  - An explicit base record specification, if any,
  - Any interfaces that the record implements, and
  - Type parameter constraints, if needed, followed by
- A semicolon `;`, or
- A record body, which consists of
  - A list of member declarations within the delimiters `{}`, and
  - An optional semicolon `;`.

Record parameters can use `in` and `params` modifiers. But, `out` and `ref` modifiers (as well as `this`-kind modifiers for extension methods) are not allowed. All positional properties are immutable. A positional record declaration, `record Team(string Country, int Rank)`, for example, is more or less equivalent to the following:

```
record class Team {
    public string Country { get; init; } = default!;
    public int Rank { get; init; } = default!;
}
```

20.2. Inheritance

Classes can only inherit from classes and records can only inherit from records (other than `object`). Records can also be `sealed` to prevent further derivation.

A derived record declares positional parameters for all parameters in the base record primary constructor. The base record declares and initializes those properties. The derived record does not hide them, but only creates and initializes properties for the parameters that are not declared in its base record. Here's a simple example:

```
abstract record Person(string FirstName, string LastName);
```

```
record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);             ①
```

① Record `Teacher` inherits from the abstract record `Person`. `Teacher` includes an init-only positional property `Grade`, but not `FirstName` and `LastName`. These two properties are inherited from `Person`.

20.3. Primary Constructors

A record type which has at least one positional parameter has a public constructor whose signature corresponds to those parameters. This is called the *primary constructor* for the type, and it hides the implicitly declared default constructor, if present. When invoked through the `new` operator, the primary constructor

- Executes the instance initializers appearing in the record body, and
- Invokes the base class constructor with the arguments provided in the base record clause, if specified.

If a record has a primary constructor with more than one positional parameters, then any user-defined constructor, except for a copy constructors, must have an explicit `this` constructor initializer.

20.4. Immutability

A positional record automatically declares init-only properties, based on the positional parameters, which are immutable. Compiler-generated methods are implemented such that they preserve the record's immutability.

Read-only properties, whether created from positional parameters or explicitly specified via readonly or init-only accessors, have *shallow immutability*. After initialization, you cannot change the values of these properties. However, the data that a reference-type property refers to can still change, e.g., via other references pointing to the same data.

20.5. Value Equality

In general, the kind of type you declare governs how equality is defined unless you override or replace equality methods:

- For class types, two objects are equal if they refer to the same object.
- For struct types, two objects are equal if they are of the same type and store the same values. This relies on reflection on run time.
- For record types, two objects are equal if they are of the same type and store the same values. This relies on the compiler-synthesized methods based on the declared data members.

20.6. Record Deconstruction

For records with two or more positional parameters, the compiler produces a `Deconstruct` method. The `Deconstruct` method has `out` parameters that match the names of all public properties in the record type, including all positional properties. The `Deconstruct` method can be used to deconstruct the record into its component properties.

When a record has a base record, the `Deconstruct` method includes all positional parameters including those inherited from the base record.

20.7. Non-Destructive Mutations

Record classes and record structs support `with` expressions. A `with` expression with a record-type argument instructs the compiler to create a copy of the given record, but *with* specified properties modified. You use the object initializer syntax to specify the values to be changed. The `with` expression can set positional properties or the `init` or `set` accessor properties. The result of a `with` expression is a *shallow copy*.

You can also use `with` expressions to create a (shallow) copy of a record. For example,

```
var copied = TemperatureRecord with { };
```

Chapter 21. Structs

In C#, value types like `bool`, `char`, and `double` can be created using *structs*. Unlike these simple types, however, structs are compound data types that can contain data members and function members.

Structs, or structures, are rather similar to classes and record classes in the way they are declared and used, with some crucial differences. For example, a variable of a struct type directly stores the data of the struct, whereas a variable of a class/record type stores a reference to a dynamically allocated object on the heap. In particular, with the exceptions of `in`, `out`, and `ref` parameters, it is not possible to create references to structs.

Furthermore, struct types do not support inheritance, and all struct types implicitly inherit from type `System.ValueType`. Here's a simple example of `struct`.

```
struct Point {
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Generally speaking, copying the entire value of a (large) struct is less efficient than copying an object reference. That is, assignment, parameter passing, and returning values can be more expensive with structs than with classes or other reference types.

On the flip side, since the variables of a struct type have their own copies of the data, it is not possible for operations on one to affect the others. In case of classes or records, which uses reference semantics, an operation on one variable can potentially affect the same object referenced by other variables. This can cause some unexpected, and possibly undesirable, behavior, especially in a multi-threading environment. For example,

```
var a = new Point(0, 5);                    ①
var b = a;                                  ②
a.x = 100;                                  ③
Console.WriteLine(b.x);                     ④
```

① A value is created through the `new` operator.

② The assignment of a value creates a new and separate copy. `b` is a `Point` struct which has the same value as value as `a`.

③ Updating the value of `a` does not affect the value of `b`. If `Point` were a class, then `a` and `b` would have pointed to the same object in memory and it would have ended up changing the member value `x`.

④ This will print out `0`, not `100`, as expected.

21.1. Struct Declarations

A struct declaration statement comprises, in the following order:

- A header including
  - Any attributes, and any struct modifiers such as `new`, `readonly`, `ref`, and `record`, including access modifiers,
  - The keyword `struct` or `partial struct`,
  - The name of the struct,
  - A list of any type parameters, if it is a generic declaration,
  - An optional positional parameter list,
  - Specification of any interfaces that the struct implements, and

    - Any type parameter constraints, if needed for any of the type parameters, followed by
- A struct body, which consists of
    - A list of member declarations within curly braces `{}`.

A struct declaration is largely similar to that of a class or record, and the meanings of each corresponding component are more or less the same. One of the biggest differences is that a struct declaration cannot specify an explicit base type. All structs inherit from `ValueType`, which in turn inherits from `Object`. On the other hand, a struct can directly implement one or more interface types.

Another major difference from the class declaration is that a struct can be declared as `readonly`. A readonly struct type is immutable.

- All data members of a `readonly struct` must be read-only:
    - Any field declaration must have the `readonly` modifier.
    - Any property must be read-only, either readonly or init-only.
- All other instance members, with the exception of constructors, are implicitly readonly.

A struct cannot be declared as `static`. Record structs and)) are discussed later in this chapter, along with ref structs and readonly ref structs. The `new` modifier has the same semantics as in the `class` declaration, and it can only be used with nested struct types.

21.2. Struct Members

21.2.1. Struct constructors

Structs are created by invoking struct constructors with the `new` operator. However, unlike in the case of reference types, the `new` expression does not necessarily allocate dynamic memory on the heap.

Instead, a struct constructor simply returns the struct value itself, e.g., created on the stack.

As of C# 10 and C# 11, there have been some big improvements on the way the struct-type values are created. In many ways, struct constructors are now more in line with class constructors. In addition, structs now support the positional record constructor syntax, using the `record` modifier, which was initially introduced for record classes.

Public parameterless instance constructors, and instance field and property initializers, can now be included in struct declarations, across all different struct kinds, `struct`, `readonly struct`, `ref struct`, `readonly ref struct`, `record struct`, and `readonly record struct`. If no parameterless instance constructor is declared, one is implicitly provided by the compiler which returns the value that results from setting all fields to their zero values.

21.2.2. Auto default structs

Beginning with C# 11, any fields that are not explicitly assigned by the user in struct constructors are automatically initialized with their `default` values.

21.2.3. Struct members

The members of a struct are the members declared in the struct in addition to the inherited members from `ValueType` and `Object`. A struct can include constants, fields, properties, methods, events, indexers, operators, other nested types, as well as static and instance constructors. Finalizers are not supported in structs, however.

21.2.4. Readonly instance members

As a generalization of the `readonly struct`, any (non-readonly) struct can include `readonly` instance members.

21.3. Ref Structs

A `ref struct`, or `readonly ref struct`, cannot be boxed. Once created, it remains on the stack throughout its lifetime. A `ref struct` can't be the element type of an array. A `ref struct` variable can't be captured by a lambda expression or a local function. A `ref struct` variable can't be used in iterators. In addition, ref structs cannot implement any interfaces. A `ref struct` can be made disposable (e.g., in the context of the `using` statement) by including an accessible `void Dispose()` method.

Furthermore, as of C# 11, you can declare a `ref` or `ref readonly` field in a `ref` struct.

21.4. Record Structs

Value type records can be created using the `record struct` or `readonly record struct` declarations. Record struct types are value types, like other struct types. Record structs cannot be declared as `ref` structs. Record struct parameters can use `in` modifier, but not `out` or `ref`. The `params` array parameter is permitted.

Positional properties are immutable in record classes and readonly record structs, but they are mutable in record structs.

```
record struct Grade(string Name, decimal Score);
```

The `Grade` struct includes two read-write properties, `Name` and `Score`.

```
readonly record struct Secret(string User, string Pass);
```

The `Secret` struct includes two init-only properties, `User` and `Pass`.

Chapter 22. Extension Methods

An extension method is a static method

- Which is defined in a non-generic, non-nested static class, and
- Whose first parameter includes modifiers `this`, `in this`, or `ref this`, but no others. (Semantics of `in` and `ref` modifiers are described in the formal parameters chapter.)

An extension method may be invoked using the instance method invocation syntax, with the receiver expression as the first argument.

22.1. `this` Extension Methods

`this T self`

`T` can be any type, either value types or reference types, or generic type parameters.

This type of extension methods use the default call-by-value semantics. In case of value types, the values are copied. In case of reference types, the references are copied, but they point to the same object. Here's an example value type:

```
record struct Tick(int Seconds);          ①
```

① We deliberately use a mutable type for illustration.

Here's a `this` type extension method:

```
static class ExtensionV1 {
    public static int TickThis(this Tick self) =>
++self.Seconds;
}
```

Here’s a simple demo program:

```
var tick = new Tick(10);
var (s0, s1, s2) = (tick.Seconds, tick.TickThis(),
tick.Seconds);
Console.WriteLine($"{s0}, {s1}, {s2}");
```

This will print out *10*, *11*, *10*. This is because `Tick` is a value type. What happens inside the extension method does not affect its value outside the method.

Here’s the same example with a `class`, which is a reference type:

```
record class Tock(int seconds) {
    public int Seconds { get; set; } = seconds;
}
static class ExtensionR {
    public static int TockThis(this Tock self) =>
++self.Seconds;
}
var tock = new Tock(10);
var (t0, t1, t2) = (tock.Seconds, tock.TockThis(),
tock.Seconds);
Console.WriteLine($"{t0}, {t1}, {t2}");
```

This program will print out *10*, *11*, *11*. This is because `Tock` is a reference type. Pass-by-value semantics have different implications for reference types.

22.2. `ref this` Extension Methods

`ref this T self`

`T` must be a struct type or a generic type parameter constrained to be a `struct`.

The `ref` parameters use pass-by-reference semantics. Here's an example, using the same mutable value type, `Tick`,

```
static class ExtensionV2 {
    public static int TickRefThis(ref this Tick self) =>
++self.Seconds;
}
var tick = new Tick(10);
var (r0, r1, r2) = (tick.Seconds, tick.TickRefThis(),
tick.Seconds);
Console.WriteLine($"{r0}, {r1}, {r2}");
```

The output of this program will be *10, 11, 11*. Similar to the reference type `Tock`, the extension method with `ref this` parameter ends up modifying the original value of the value `tick`.

22.3. `in this` Extension Methods

`in this T self`

`T` must be an actual struct type.

Here's an example of the `in this` extension methods:

```
static class ExtensionV3 {
    public static int TickInThis(in this Tick self) =>
self.Seconds;
}
```

The `in` modifier has the same effect as the `ref` modifier. Both make the modified parameter use pass-by-reference semantics. But, unlike the `ref this` parameter, the `in this` parameter cannot be modified. When we call this extension method, e.g., `tick.TickInThis()`, the same `tick` value is passed into the method. There is no copy.

Chapter 23. The `new` Operator

A `new` expression starts with the keyword `new`, and it returns a new instance of an (explicitly or implicitly) specified type. The `new` operator is used with a few different categories of types:

- Arrays,
- Objects of class, record, and struct types, with
  - Object initializers,
  - Collection initializers, and
- Delegates.

23.1. New Array Expressions

An array object can be created in a number of different ways:

```
int[] a1 = new int[3];                 ①
int[] a2 = new int[3] { 1, 2, 3 };     ②
int[] a3 = new int[] { 1, 2, 3 };      ③
int[] a4 = new[] { 1, 2, 3 };          ④
int[] a5 = { 1, 2, 3 };                ⑤
```

① This creates an `int` array of three elements, with the default value (`0` for `int`) for each element.

② This creates an `int` array of three elements, with the given values.

③ The same as above. The number of elements can be inferred from the initializer expression, `{ 1, 2, 3 }`.

④ The same. The element type can be inferred from the initializer, or from the explicitly specified type `int[]`.

⑤ The same. This is the most succinct syntax, but this syntax cannot be used with the implicit variable declaration using `var`.

23.2. New Object Expressions

The `new` object expression has the following general syntax.

```
new TYPE INITIALIZER
new TYPE ( ARGUMENT_LIST ) INITIALIZER
```

`INITIALIZER` can be either an object initializer or a collection initializer. Note that an object initializer may contain collection objects with collection initializer syntax, and vice versa.

23.3. Object Initializers

An object initializer consists of a sequence of member initializers, enclosed in `{` and `}` and separated by commas ','.

```
{ MEMBER_INITIALIZER, MEMBER_INITIALIZER, }
```

`MEMBER_INITIALIZER` can be one of the following four forms. The square bracket syntax is used for indexers.

```
IDENTIFIER = EXPRESSION
IDENTIFIER = INITIALIZER
[ ARGUMENT_LIST ] = EXPRESSION
[ ARGUMENT_LIST ] = INITIALIZER
```

For example,

```
public record Ball(int Size = 0) {                        ①
    public required string Sports { get; init; }          ②
    private int magic = 1;
    public int this[int index] {                          ③
```

```
        get => magic;
        set => magic = value;
    }
}
Ball ball1 = new Ball() { Sports = "Soccer" };          ④
Ball ball2 = new() { Sports = "Football", Size = 10 }; ⑤
Ball ball3 = new Ball(20) {                             ⑥
    Sports = "Ping Pong",
    [0] = 2,
};
Ball ball4 = new(30) {                                  ⑦
    Sports = "Tennis",
    [0] = 3,
    [1] = 30,
};
```

① This record class has a primary constructor with an optional parameter, `Size`. `Size` is also a public init-only property.

② The `required` property need to be specified in the initializer.

③ An indexer, with a dummy implementation.

④ One initializer syntax, `new` + constructor + object initializer.

⑤ Target typed `new()` + object initializer.

⑥ Another initializer syntax, `new` + constructor + object initializer.

⑦ Target typed `new` + `(` constructor arguments `)` + object initializer.

23.4. Collection Initializers

Here are a few different collection initializer examples:

```
List<int> list1 = new List<int> { 1, 2, 3 };
List<int> list2 = new() { 10, 20, 30 };
Dictionary<int, string> dictionary1 = new Dictionary<int,
string> {
    [3] = "tres",
    [5] = "sinco",
};
Dictionary<int, string> dictionary2 = new() {
    [8] = "ocho",
    [10] = "diez",
};
```

23.5. Anonymous Object Initializers

The `new` operator can also be used to create an object of an *anonymous type*. An anonymous object initializer implicitly declares an anonymous type and it returns an instance of that type. An anonymous type is a nameless class type that inherits directly from `object`. For example,

```
var sorceror = new { Rank = 99, Name = "Gandalf the White" };
Console.WriteLine(sorceror);
// { Rank = 99, Name = Gandalf the White }
```

The anonymous object initializer in this example declares an anonymous type which has two readonly properties, `Rank` of type `int` and `Name` of type `string`, and it creates an instance with the given values, `99` and `"Gandalf the White"`, respectively.

23.6. New Delegate Expressions

Similarly, a delegate instance can be created using the following syntax:

```
new DELEGATE_TYPE ( EXPRESSION )
```

Chapter 24. Expressions

A C# expression comprises a sequence of operators and operands and it evaluates to a value according to the prescribed rules.

There are three kinds of operators, the unary operators, binary operators, and ternary operators, which take one, two, and three operands, respectively. C# includes most commonly-used operators for arithmetic, comparison, and boolean logic, etc. Moreover, C# defines many additional operators for various purposes such as `typeof`, `await`, and `with` operators. Some of them are described in this chapter, and some others are described throughout the book. Many of the C# operators can be "overloaded" for custom types, as discussed earlier.

The order of evaluation of operators in an expression is determined by the *precedence* and *associativity* of the operators The operands in an operation are normally evaluated from left to right. The result of an expression can be a variable, a value, a property or index access (which is reclassified as a value), an anonymous function, a `null` literal, or "nothing" (`void`).

24.1. Expression Statements

Certain kinds of expressions, called the *statement expressions*, can also be used as standalone statements. For example,

- Method invocation expressions,
- Object creation using the `new` operator,
- Assignments (both simple and compound assignments),
- Increment/decrement expressions (both prefix and postfix),
- Await expressions, and
- Throw expressions.

Execution of an *expression statement* evaluates the expression, discards the evaluated value, and then transfers control to the end point of the expression statement.

24.2. The `checked` and `unchecked` Statements

The `checked` and `unchecked` statements are used to control the overflow checking context for integral-type arithmetic operations and conversions.

- In a checked context, an overflow produces a compile-time error or throws a `System.OverflowException`.
- In an unchecked context, overflows are ignored and any high-order bits that do not fit in the destination type are discarded. This is the default behavior in C#.

```
checked {
    Console.WriteLine(int.MaxValue + 1);       ①
}
unchecked {
    Console.WriteLine(int.MaxValue + 1);       ②
}
```

① This checked statement will not compile.

② This overflows at run time, and it will likely print out `-2147483648`.

24.3. Classifications of C# Expressions

Most C# operators can be *overloaded*. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined type. C# operators can be categorized as follows.

- Arithmetic operators,
- Comparison operators,
- Assignment operators,
- Boolean logic operators,
- Bitwise operators,
- Ternary operator,
- Default value operator,
- `true` and `false` operators,
- `null` forgiving operator,
- `null` coalescing operator,
- Type-testing operators,
- `sizeof` operator,
- `nameof` operator,
- `typeof` operator,
- `as` operator,
- `is` operator, and
- `switch` expressions.

These operators are explained in this chapter. In addition, the following operators and expressions are described, or briefly mentioned, throughout the book.

- Member access operator,
- `await` operator,
- `new` operator,
- `with` expressions,
- Lambda expressions,

- local functions,
- Throw expressions, and
- Patterns.

We do not discuss the operators that are only allowed in the unsafe code such as `stackalloc` and other pointer-related operators.

24.4. The Operator Precedence

When an expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated. For example, the expression `1 + 2 * 3` is evaluated as `1 + (2 * 3)`, which results in `7`. This is because the multiplication operator `*` has higher precedence than the addition operator `+`.

24.5. The `checked` Operators

Since C# 11, the following operators can be optionally declared as `checked` when overloaded:

- The `++` and `--` unary operators,
- The `-` unary operator,
- The `+`, `-`, `*`, and `/` binary operators, and
- Explicit conversion operators.

24.6. Arithmetic Operators

24.6.1. Unary plus operator

```
T operator +(T x);
```

The unary plus operator is predefined for `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` types.

24.6.2. Unary minus operator

```
T operator -(T x);
```

The unary minus operator is defined for `int`, `long`, `float`, `double`, and `decimal` types.

24.6.3. Prefix increment and prefix decrement

```
var x = 0;
++x;                       // Prefix increment
--x;                       // Prefix decrement
```

24.6.4. Postfix increment and postfix decrement

```
var x = 0;
x++;                       // Postfix increment
x--;                       // Prefix decrement
```

24.6.5. Multiplication operator

```
T operator *(T x, T y);
```

The multiplication operator is defined for `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` types.

24.6.6. Division operator

```
T operator /(T x, T y);
```

The division operator is defined for `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` types.

24.6.7. Remainder operator

```
T operator %(T x, T y);
```

The remainder operator is defined for `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` types.

24.6.8. Addition operator

```
T operator +(T x, T y);
```

The addition operator is defined for `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` types, as well as for enums, delegates, and string types.

24.6.9. Subtraction operator

```
T operator -(T x, T y);
```

The subtraction operator is defined for `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` types, as well as for enum and delegate types.

24.7. Relational Operators

```
x == y  true if x is equal to y, false otherwise
x != y  true if x is not equal to y, false otherwise
x < y   true if x is less than y, false otherwise
x > y   true if x is greater than y, false otherwise
x <= y  true if x is less than or equal to y, false otherwise
x >= y  true if x is greater than or equal to y, false
otherwise
```

24.8. Assignment Operators

The assignment operators assign a new value to a variable, a property, an event, or an indexer element. They are right-associative.

Simple assignment operator

`=`

It assigns the value of the right operand to the variable, property, or indexer element given by the left operand.

Compound assignment operators

`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`

It performs the indicated operation on the two operands, and then assign the resulting value to the variable, property, or indexer element given by the left operand.

Event assignment operators

`+=`, `-=`

An event access can be only used on the left hand side of assignment with these two operators.

Null-coalescing assignment operator

`??=`

It assigns the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to `null`.

24.9. Logical Operators

24.9.1. Logical operators

The logical operators, AND (`&`), OR (`|`), and exclusive OR (`^`), are overloaded and they can be used among integer values, enum values, and bool values.

Boolean Operands

- `true & true` returns `true`. `x & y` for All other combinations of operand values yields `false`.
- `false | false` returns `false`. Otherwise, `x | y` for all other pairs of `x` and `y` yields `true`.
- Either `true ^ false` or `false ^ true` returns `true`. Otherwise, `x ^ y` is `false`.

Nullable Boolean Operands

The logical AND `&` and OR `|` operators are also predefined between nullable bool values. The `null` value more or less acts like `false` except that `true & null`, `null & true`, `null & null`, `false | null`, `null | false`, and `null | null` evaluate to `null`.

Integer Operands

The `&` operator computes the bitwise logical AND, the `|` operator computes the bitwise logical OR, and the `^` operator computes the bitwise logical exclusive OR of the two operands.

Enum Operands

These three logical operators all also predefined for all enum types. Their values are computed by

1. First converting the operands to their underlying types,
2. Computing the result of the given logical operation, and
3. Converting back the result to the original enum type.

24.9.2. Conditional logical operators

Conditional logical operators, `&&` and `||`, in C# evaluate their first operands first, and based on those values, they may or may not evaluate their second operands. This is often known as "short circuiting".

- The logical AND operation `x && y` corresponds to the operation `x & y`, except that `y` is evaluated only if `x` is not `false`.
- The logical OR operation `x || y` corresponds to the operation `x | y`, except that `y` is evaluated only if `x` is not `true`.

24.10. Bitwise Operators

The bitwise logical AND `&`, OR `|`, and exclusive OR `^` operators are included in the previous section.

24.10.1. Logical negation operator

The logical negation operator (`!`) is predefined as follows:

```
bool operator !(bool x);
```

This operator computes the logical negation of the operand. That is, if the operand is `true`, it returns `false`, and vice versa.

24.10.2. Bitwise complement operator

The bitwise complement operator (`~`) is predefined for certain integer types, e.g., `int`, `uint`, `long`, and `ulong`. For example,

```
int operator ~(int x);
```

Every enum type `E` implicitly provides the following bitwise complement operator:

```
E operator ~(E x);
```

The result of evaluating `~x`, where `x` is an expression of an enum type `E` with an underlying type `U`, is exactly the same as evaluating` (E)(~(U)x)`.

24.11. Shift operators

There are three kinds of binary bit shifting operators, as of C# 11:

- Left shift (`<<`),
- Right shift (`>>`), and
- Unsigned right shift (`>>>`) operators.

They are predefined for `int`, `uint`, `long`, and `ulong` types. When declaring an overloaded shift operator for a class, record, or struct,

- The first operand must be the given type, and
- The second operand is always of the `int` type.

For any of the predefined operators, the number of bits to shift in `x SHIFT n` is computed as follows:

- When x is int or uint, the shift count is given by the low-order five bits of n. That is, the shift count is the same as n & 0x1F.
- When x is long or ulong, the shift count is given by the low-order six bits of n, which is equal to n & 0x3F.

The >>> operator shift bits right without replicating the high order bit on each shift. In case of primitive integer types, this is equivalent to casting an integer to the comparable unsigned type, shifting using a regular right shift operator, and casting the result back to the original type, if necessary.

For example,

```
0b11111 << 2
0b11111 >> 3
-0b11111 >>> 3
```

24.12. True and False Operators

A pair of operators, true and false, can be used to customize the way instances of a type evaluates in Boolean expressions. The pair should be implemented together, and they are a generalization of the implicit bool conversion.

The true operator is to return true if the operand is "definitely true". Otherwise, it returns false, that is, if it is false or if it can be true or false, etc. Likewise, the false operator is to return true if and only if the operand is "definitely false".

For example,

```
public static bool operator true(T x);
public static bool operator false(T x);
```

24.13. Conditional Operator

The ternary conditional operator `?:` has the following two forms:

```
EXP_B ? EXP_X : EXP_Y
ref (EXP_B ? ref EXP_X : ref EXP_Y)
```

The first operand of the `?:` operator `EXP_B` is

- Either an expression that can be implicitly converted to `bool`, or
- An expression of a type that implements operator `true`.

The conditional expression first evaluates `EXP_B`:

- If the result is `true`, `EXP_X` is evaluated, and its value, or its `ref`, becomes the result of the whole expression.
- Otherwise, `EXP_Y` is evaluated, and its value, or its `ref`, is returned as the result.

Note that the second and third operands, `EXP_X` and `EXP_Y`, determines the type of the conditional expression. The ternary operator is right-associative.

The second form of the conditional expression produces a `ref` result, and it can be assigned to a `ref` variable. For example,

```
int? a = 10;
int? b = 20;
ref var c = ref (a != null ? ref a : ref b);

c = 100;
Console.WriteLine($"a = {a}, b = {b}");  ①
```

① The output will be *a = 100, b = 20.*

24.14. The Index and Range Operators

We can index and slice a sequence at runtime using `System.Index` and `System.Range` objects. An array is a prime example that supports both indexing and slicing operations.

24.14.1. `System.Index`

`Index` is a readonly struct that can be used to index a collection either from the beginning or from the end. It implements `IEquatable<Index>`.

To use the `Index` type as an argument in an array element access, the following member is required:

```
int System.Index.GetOffset(int length);
```

The hat `^` operator

To use the hat operator `^`, that is, indexing from the end, the following member is required:

```
public System.Index.Index(int value, bool fromEnd);
```

24.14.2. `System.Range`

`Range` is a readonly struct representing a range that has both start and end indexes, implicitly or explicitly. It also implements `IEquatable<Index>`.

The `..` syntax allows for either, both, or none of its arguments to be absent. The `..` syntax for `Range` will require the `Range` type, as well as one or more of the following members:

```
public System.Range.Range(System.Index start, System.Index
end);
public System.Range.static Range StartAt(System.Index start);
public System.Range.static Range EndAt(System.Index end);
public System.Range.static Range All { get; }
```

In addition, if the following member is present a value of type `Range` can be used in an array element access expression:

```
public static T[]
System.Runtime.CompilerServices.RuntimeHelpers.GetSubArray<T>(
T[] array, System.Range range);
```

24.15. Default Value Operator

A default value expression is used to obtain the default value of a type. There are two forms.

Explicitly Typed Default

```
default ( TYPE )
```

The `default` operator takes a parenthesized type as an operand, and it returns the default value of the specified type.

```
Func<string, bool> whereClause = default(Func<string,
bool>);
```

Default Literal

```
default
```

The literal `default`, representing a default value, is untyped. It can be converted to any type through a default literal conversion.

```
Func<string, bool> whereClause = default;
```

24.16. Null-Testing Expressions

Null conditional expressions can be used with potentially nullable values and references, e.g., to avoid null reference exceptions.

24.16.1. Null conditional member access

A null conditional member access uses two tokens `?.` instead of the normal single token `.`. Given a member access expression of the form `P.A` of the type `T`, `P?.A` evaluates as follows, depending on whether `T` is a (non-nullable) value type or a reference type.

When `T` is a value type

```
((object)P == null) ? (T?)null : P.A
```

The type of `P?.A` is `T?`.

When `T` is a reference type

```
((object)P == null) ? null : P.A
```

The type of `P?.A` is `T`, the same as that of `P.A`.

That is, if `P` is `null`, it returns `null`. Otherwise it returns the value of the normal member access expression `P.A`.

24.16.2. Null conditional element access

A null conditional element access uses the syntax `?[…]` instead of the normal subscription syntax `[…]`. Given a member access expression of the form `P[A]` of the type `T`, `P?[A]` evaluates as follows, depending on whether `T` is a (non-nullable) value type or a reference type.

When `T` is a value type

```
((object)P == null) ? (T?)null : P[A]
```

The type of `P?[A]` is `T?`.

When `T` is a reference type

```
((object)P == null) ? null : P[A]
```

The type of `P?[A]` is `T`, the same as that of `P[A]`.

That is, if `P` is `null`, it returns `null`. Otherwise it returns the value of the normal element access expression `P[A]`.

24.16.3. Null conditional invocation expression

A null conditional invocation is syntactically a normal invocation expression preceded by null conditional member or element access.

24.16.4. The null coalescing operator

A null coalescing expression `A ?? B` can be viewed as a syntactic short form of a conditional expression `X ? Y : Z` in certain circumstances. In particular, `A ?? B` is more or less equivalent to `(A != null) ? A : B`. That is, if `A` is non-null, the result is `A`. Otherwise, the result is `B`. There is also the null-coalescing assignment operator `??=`, which is a compound assignment operator with `??`.

24.16.5. Null forgiving operator (postfix `!`)

The unary postfix `!` operator is the null-forgiving, or null-suppression, operator. It is used to suppress all nullable warnings for the preceding expression. The null-forgiving operator has no effect at run time. Expressions `x!` and `x!.m` evaluate to the results of the underlying expressions `x` and `x.m`, respectively.

24.17. The `nameof` Operator

A `nameof` expression returns the *name* of a given program entity as a constant string, evaluated at compile time. The program entity itself is not evaluated, and the `nameof` operator does not affect the run time behavior. For example,

```
var fruit = "avocado";
Console.WriteLine($"{nameof(fruit)} = {fruit}");
```

It should be noted that the names of the program entities like variables and methods, etc. are not significant in program execution. That is, it has no consequences whether you name a particular variable `sum` or `added` or `xyz`, etc., as long as you use the name consistently. The `nameof` operator is an exception, and it is primarily used for diagnostic purposes during development.

One other special behavior of the `nameof` operator is that, in certain cases, it can use a name beyond the name's normal scope. In particular, `nameof`, when used in an attribute, can reference formal parameters or type parameters of the decorated method. For example,

```
[MyAttribute(nameof(myParam))] void MyMeth(int myParam) { }
[MyAttribute(nameof(TypeParam))] void MyMeth<TypeParam>() { }
```

24.18. The `sizeof` Operator

The `sizeof` operator with a certain type operand returns the number of bytes that a variable of the given type is supposed to occupy.

```
sizeof ( VALUE_TYPE )
```

For some predefined types, the `sizeof` operator yields a constant `int` value. For example, `sizeof(ulong)` is `8` since `ulong` is a 64-bit (8-byte) integer type. For an enum type `E`, the result of the expression `sizeof(E)` is a constant value equal to the size of its underlying type.

24.19. The `typeof` Operator

The `typeof` operator is used to obtain the `System.Type` object for a type. It can be used with an operand of any non-dynamic type `TYPE` or the keyword `void`, in a pair of parentheses. For example,

```
typeof ( TYPE )
typeof ( void )
```

There is only one `Type` object for any given type. That is, for a type `T`, `typeof(T) == typeof(T)` always yields `true`. The value of `typeof(void)` is distinct from those of `typeof(T)` for any type `T`. It represents the absence of a type.

24.20. The `as` Operator

The `as` operator explicitly converts the result of an expression to a given reference or nullable value type. If the conversion isn't possible, the `as` operator returns `null`. The expression of the form `E as T` produces the same result as `E is T ? (T)(E) : (T)null`.

24.21. The is Expression

The is operator has two uses in C#. First, it is used to check if the run-time type of a given expression is compatible with a given type, e.g., E is T. It returns true if the result of E is the type T, or if it is not null and it is convertible to T via a reference conversion, a boxing or unboxing conversion, or a wrapping or unwrapping conversion.

24.21.1. Pattern matching

The is operator can also be used to match a given expression E against a given pattern P, e.g., E is P. This is known as pattern matching. This use of the is expression also returns a Boolean value. For example,

```
if(marriage is not null) {
    Console.WriteLine("Huh???");
}
if(fruit is Apple apple) {
    Console.WriteLine($"I am an apple, {apple}.");
}
if(list is [1, 2, ..]) {                ①
    Console.WriteLine("The list starts with 1 and 2.");
}
```

① The right hand side of the is operator, [1, 2, ..], is a list pattern. This pattern is available since C# 11.

24.22. The switch Expression

The switch expression is similar to the switch statement, and it also uses pattern matching. Compared to the switch statement, the switch expression has a few syntactic improvements:

- The overall structure is, a target expression and the switch operator followed by a switch block.

- It uses the fat arrow token `=>` in place of the `case:` elements in the `switch` statement.
- It uses the discard variable _ for the `default` case.
- The body of each pattern-based branch is an expression, and not a statement.

```
static float AvgTemp(Month month) => month switch {  ①
    Month.June => 70f,                               ②
    Month.July => 80f,
    Month.August => 90f,
    _ => float.NaN,                                  ③
};

enum Month : byte {
    June = 6,
    July,
    August,
}
```

① It is a common pattern to use an expression body with a `switch` expression. Note that the target expression `month` comes before the `switch` keyword unlike in the `switch` statement.

② Each branch has a form, "PATTERN => EXPRESSION", and these branches are separated by commas. This syntax is more concise and intuitive than its counterpart in the `switch` statement, which tends to require repetitive uses of `case` and `break` statements. When a pattern matches, the value of its corresponding expression, e.g., `70f` in this case, becomes the value of the overall `switch` expression.

③ The discard variable here _ is called the discard pattern, and it corresponds to the `default` case. Various pattern types are further described later in the Pattern Matching chapter.

Chapter 25. Lambda Expressions

A lambda expression is an anonymous function definition that can be used "in place" where a method or a delegate is expected. That is, a lambda function requires no separate declaration and subsequent call(s), in two steps.

A lambda expression has a natural type, and it is compatible with a delegate or expression tree type. In general, its type can be easily inferred. Although it is syntactically an expression, it can also include a block of statements. Lambda expressions are closures, and they are often used with query expressions (e.g., LINQ).

25.1. Lambda Functions

Syntactically, there are two kinds of lambda expressions, an "expression lambda" and a "statement lambda".

25.1.1. Expression lambdas

A Lambda expression is defined with a Lambda declaration operator `=>`. On the left hand side, a function parameter list is specified. On the right hand side comes an expression in case of expression lambdas.

```
( PARAMETERS ) => EXPRESSION
```

For example, `(int x, int y) => x + y` is a lambda function. (This anonymous function can be cast to the type of `Func<int, int, int>`, which is a .NET delegate type that takes two `int` arguments and returns an `int` value. We do not discuss `Func` and `Action` types in this book.)

25.1.2. Statement lambdas

One can also use a block on the right hand side of a lambda declaration operator `=>`. Such expressions are called statement lambdas.

```
( PARAMETERS ) => { STATEMENTS }
```

The last statement of `STATEMENTS` can be a return statement.

25.2. Static Lambda Functions

Static lambda expressions, with the `static` modifier, are analogous to the `static` local functions. A static lambda or static anonymous method can't capture local variables or other instance state.

```
static ( PARAMETERS ) => EXPRESSION
static ( PARAMETERS ) => { STATEMENTS }
```

25.3. Closures

A non-static lambda expression can "capture" the variables in the outer scope. The captured, or "closed-over", variables are stored for use in the lambda expression even if they would otherwise go out of scope and potentially be garbage-collected.

For example, here's a function that prints out the first `n` numbers of Fibonacci sequence:

```
static void PrintFibonacci(int n) {                        ①
    var f = fibonacci();
    foreach (var i in Enumerable.Range(1, n)) {            ②
        Console.WriteLine($"{i}: {f()}");                  ③
    }
    static Func<int> fibonacci() {                        ④
        var (a, b) = (0, 1);                              ⑤
        return () => {                                    ⑥
            (a, b) = (b, a + b);                          ⑦
            return a;
        };
    }
}
```

① In the context of the top-level statement, this function is a static local function in the implicitly generated `Main` method.

② The `Enumerable.Range` static method can be used to generate a simple integer sequence.

③ We call the function `f`, which is returned by `fibonacci()`, multiple times to see the effect of the captured variables.

④ A local static function declaration within the `PrintFibonacci` function.

⑤ `a` and `b` are local variables of `int` type.

⑥ A lambda function. Note that this is a statement lambda, which takes no argument and returns an `int` value. That is, its type is `Func<int>`.

⑦ In a non-static lambda function, the outside variables can be used, and these variables are closed-over after the function has been called and returned.

Chapter 26. Statements

Statements control execution of a program. C# supports most of the common statements found in other C/C++-style programming languages. For the sake of completeness, we go through some of the C# statements in this chapter. Statements in C# can be classified into a few different categories.

Declaration Statements

A declaration statement is used to declare new local variables and constants and optionally initialize them.

Iteration Statements

An iteration statement is used to repeatedly execute an embedded statement, e.g., a block of statements.

- The `for` statement: Executes its body while a specified Boolean expression evaluates to `true`.
- The `foreach` statement: Enumerates the elements of a collection and executes its body for each element of the given collection.
- The `do-while` statement: Conditionally executes its body, but at least once.
- The `while` Statement: Conditionally executes its body zero or more times.

Selection Statements

A selection statement is used to select one of a number of possible execution branches based on the value of a given expression.

- The `if` statement: Selectively executes a particular block of statements based on the value of a Boolean expression.
- The `switch` statement: Selectively executes a block of statements based on a pattern match with a given match expression.

Jump Statements

A jump statement is used to transfer control. In this group are the `goto`, `break`, `continue`, `return`, `throw`, and `yield` statements.

- The `goto` statement: Transfers control to a statement with a target label.
- The `break` statement: Terminates the innermost iteration statement or `switch` statement.
- The `continue` statement: Finishes the current iteration of the innermost iteration statement.
- The `return` statement: Terminates execution of the function in which it appears and returns control to the caller.
- The `ref return` statement: Terminates execution of the function. It returns the result expression by reference to the caller, and not by value.

Lock Statements

The `lock` statement acquires the mutual-exclusion lock (or, `mutex`) for a given object, executes a statement block, and then releases the lock.

In addition, a block is syntactically a statement. Some expressions can be used as expression statements. We also discuss the `using` statement, `checked` and `unchecked` statements, and `try` and `throw` statements in other chapters. The `unsafe` and `fixed` statements are only allowed in the unsafe code, and they are not included in this book.

26.1. Empty Statement

An empty statement does nothing.

```
;
```

An empty statement is used when there are no operations to perform in a context where a statement is required. For example, the following is a valid statement:

```
for(;;) ;                              ①
```

① The `for` statement syntactically requires a statement following the `for ()` clause, and an empty statement is used in this example.

26.2. Declaration Statement

A declaration of local constants, local variables, or (static or non-static) local functions is a statement. Declaration statements are permitted in blocks, but they are not permitted as embedded statements. All variables and constants in C# have declared types.

26.3. The `for` Statement

The `for` statement is a control flow statement in C# that allows you to execute a block of statements multiple times.

The `for` statement has the following syntax:

```
for ( INITIALIZER ; CONDITION ; ITERATOR ) STATEMENT
```

Or since, more commonly, a block statement is used for `STATEMENT`, the syntax appears as follows:

```
for ( INITIALIZER ; CONDITION ; ITERATOR ) {
    STATEMENTS
}
```

The `INITIALIZER` expressions, which can be zero, one, or more local variable declarations or statement expressions, separated by commas, are evaluated first. Then, if the `CONDITION` expression evaluates to `true`,

- It executes zero, one, or more `STATEMENTS`,
- It evaluates the `ITERATOR` expressions, comprising zero, one, or more statement expressions, and
- It re-evaluates `CONDITION`.

As long as the value of `CONDITION` remains `true`, it repeats the loop. Otherwise, it terminates the loop. `INITIALIZER`, `CONDITION`, and `ITERATOR` are all optional.

For example,

```
for (                                        ①
    var i = 0;                               ②
    i < 10;                                  ③
    i++                                      ④
    ) {
   Console.WriteLine(i);                     ⑤
}
```

① The `for` statement starts with the keyword `for` and ends with a block.

② The initializer clause is typically used to declare and initialize loop variables, `i` in this example.

③ The condition expression evaluates to bool.

④ In the iterator clause, we re-evaluate all expressions at each iteration.

⑤ The block can be empty. It may include `break`, `continue`, or `return` statements, etc.

26.4. The foreach Statement

The foreach statement is another iteration statement that executes an embedded statement, or a block of statements, for each element of a given collection.

```
foreach ( var IDENTIFIER in EXPRESSION ) {      ①
    STATEMENTS
}
```

① A more accurate syntax would be `foreach ( TYPE/var IDENTIFIER in EXPRESSION ) STATEMENT` where `STATEMENT` represents a (single) embedded statement. But, this syntactic notation is used informally in this book to provide some commonly used syntax. Being precise is not the goal. In this case, as with the `for` statement, a block statement is more commonly used in place of `STATEMENT`.

One can also specify a proper type of `IDENTIFIER` instead of the `var` keyword to declare this iteration or loop variable. `IDENTIFIER` corresponds to a read-only local variable with a scope that extends over the `foreach` block. During execution of a `foreach` statement, the iteration variable represents the collection element for which an iteration is currently being performed. The `EXPRESSION` should be an enumerator type.

For example,

```
string[] teams = {"AR", "HR", "FR", "MA"}; ①
foreach (var team in teams) {             ②
    Console.WriteLine(team);              ③
}
```

① The variable `teams` is a string array, which is an enumerable.

② The foreach statement starts with the keyword foreach. The block can contain zero, one, or more statements.

③ The type of the iteration variable team is string.

26.5. The do Statement

The do-while statement executes an embedded statement, or a block of statements, at least once, and possibly more depending on the given condition.

```
do {
    STATEMENTS
} while ( BOOLEAN_EXPRESSION )
```

For example,

```
string s;                          ①
do {
    s = Console.ReadLine();        ②
    if (s != null) {
        Console.WriteLine(s);
    }
} while (s != null);               ③
```

① The local variable s is not initialized, and it cannot be used to read its value until a value is set.

② When it enters the do statement, control is transferred to the statements in the block. When this statement is executed, s will be initialized, which can be null (because the Console.WriteLine method returns the type string?).

③ As long as s is not null, the statements in the block are executed again.

26.6. The `while` Statement

The `while` statement executes a block of statements as long as a given condition evaluates to true.

```
while ( BOOLEAN_EXPRESSION ) {          ①
    STATEMENTS
}
```

① This block can also be a single statement.

Note that, depending on the value of `BOOLEAN_EXPRESSION`, the statements in the block may end up being executed zero, one, or more times. For example,

```
var (i, max) = (0, 10);                 ①
while (i < max) {                       ②
    Console.WriteLine(i++);             ③
}
```

① A "multi-variable" declaration with initialization, using the tuple deconstruction syntax.

② This `while` loop start executing the block statement since `i < max` (`0 < 10`) initially.

③ At every iteration, we increase the value of `i` by `1` using the postfix increment operator. When the condition `i < max` no longer holds true, or when `i >= max` (`10 >= 10`), the loop exits.

26.7. The `if` Statement

The `if` and `switch` statements are used to select one of multiple possible groups of statements for execution based on some expressions.

The `if` statement has two forms. First,

```
if ( BOOLEAN_EXPRESSION ) STATEMENT
```

`STATEMENT` can be a block statement, which can include other statements. In fact, that is the most commonly used kind of statement in this context. Hence,

```
if ( BOOLEAN_EXPRESSION ) {
    STATEMENTS                               ①
}
```

① In this notation, `STATEMENTS` can include zero, one, or more C# statements.

In an alternative form of the `if` statement, this `if` part can be followed by the keyword `else` and another statement. That is,

```
if ( BOOLEAN_EXPRESSION ) STATEMENT else STATEMENT
```

Or, more commonly,

```
if ( BOOLEAN_EXPRESSION ) {
    STATEMENTS
} else {
    STATEMENTS
}
```

For instance,

```
// int[] arr = { 2, 1, 3, 4, 7, 11 };
```

```
if (arr.Length > 5) {                       ①
    Console.WriteLine("It's too long");
}
```

① The first form of the `if` statement.

```
if (arr.Length > 5) {                       ①
    Console.WriteLine("It's too long");
} else {
    Console.WriteLine("It's too short");
}
```

① The `if-else` form of the `if` statement.

```
if (arr.Length > 5) {                       ①
} else if (arr.Length > 2) {                ②
}
```

① The `if-else` form.

② This `else` includes a single `if` statement (not a block). This `if` statement includes a block, and it has the first form (without `else`).

```
if (arr.Length > 5) {
} else if (arr.Length > 2) {                ①
} else {                                    ②
}
```

① The same as above. This `else` includes a single `if` statement. Note that there is no separate `else if` or `elsif` in C#.

② This `else` belongs to the second `if` statement.

The `else` part is associated with the lexically nearest preceding `if` that is allowed by the syntax. Here's a somewhat convoluted example:

```
if(arr.Length > 5) if(arr.Length > 10) Console.WriteLine("It's
too loong"); else Console.WriteLine("It's just long");
```

The else belongs to the second if statement. This can be rewritten as follows, to make this fact clear:

```
if(arr.Length > 5)
    if(arr.Length > 10)
        Console.WriteLine("It's too loong");
    else
        Console.WriteLine("It's just long");
```

Regardless, it is a good practice to use a block statement ({}) instead of a single embedded statement in many of C#'s compound statements.

26.8. The switch Statement

The switch statement is another selection statement that branches based on the value of switch expression. For example, here’s a typical syntax of the switch statement.

```
switch ( EXPRESSION ) {                       ①
    case LABEL:                               ②
        STATEMENTS                            ③
    case LABEL: case LABEL:                   ④
        STATEMENTS
    default:                                  ⑤
        STATEMENTS                            ⑥
}
```

① A switch statement consists of the keyword switch, followed by a parenthesized switch expression, EXPRESSION, and a switch block, enclosed in curly braces {}. (Note that this not a block statement.)

② The switch block consists of zero or more switch sections Each switch section consists of one or more labels, `case` or `default` labels, and a list of one or more statements.

③ `STATEMENTS` needs at least one statement here. C# `switch` statement does not allow fall-through from one case label to the next, or exit the `switch` statement entirely. The last statement in `STATEMENTS` must be a jump statement that moves control out of the `switch` statement, such as a `break` statement.

④ There can be more than one labels in each section.

⑤ There can be at most one default label in a `switch` statement.

⑥ Although this is the last switch section, the last statement in `STATEMENTS` still has to be a jump statement.

For example,

```
switch (number) {
    case 0: default:
        Console.WriteLine("Don't know. Don't care.");
        break;
    case 1: case 4: case 6:
        Console.WriteLine("Non-prime");
        break;
    case 2: case 3: case 5: case 7:
        Console.WriteLine("A prime number");
        break;
}
```

26.8.1. Pattern matching

The `switch` statement is now generalized such that the case `LABEL` can be an arbitrary pattern. The traditional form of the `switch` statement can be viewed as using constant patterns as labels. Here's a simple example:

```
static Season MonthToSeason(Month month) {
    switch (month) {
        case >= Month.September:
            return Season.Winter;
        case >= Month.March:
            return Season.Summer;
        default:
            return (Season)0;
    }
}
enum Season { Summer = 2, Winter = 4, }
enum Month { March = 3, May = 5, July = 7, September = 9, }
```

Note that patterns are scanned from top to bottom. Patterns like `>= Month.September` are called the relational patterns. We can try and rewrite this `MonthToSeason` function using a `switch` expression:

```
static Season MonthToSeason(Month month) => month switch {
    >= Month.September => Season.Winter,
    >= Month.March => Season.Summer,
    _ => (Season)0,
};
```

26.9. Labeled Statements

A labeled statement permits a statement to be prefixed by a label. Labeled statements are permitted in blocks, but they are not permitted as embedded statements.

```
LABEL : STATEMENT
```

A labeled statement declares a label with the name given by `LABEL`. The scope of a label is the whole block in which the label is declared, including any nested blocks. Execution of a labeled statement corresponds exactly to the execution of the statement following the label. A label can be referenced from `goto` statements within the scope of the label.

26.10. The `goto` Statement

Jump statements such as `goto` unconditionally transfer control. The `goto` statement is the only statement in C# that can transfer control using specific target labels. The `goto` statement has the following few different forms:

```
goto LABEL;                                 ①
goto case EXPRESSION;                       ②
goto default;
```

① `LABEL` is one of the labels defined by labeled statements.

② The last two forms of the `goto` statement can be used in `switch` statements. Note that `EXPRESSION` should be a constant label, and it cannot be a general pattern.

Here’s a simple example of using `goto`:

```
var counter = 0;
Label: ++counter;
Console.WriteLine($"counter = {counter}");
if (counter < 5) {
    goto Label;
}
```

This sample code works more or less like a `for` loop. The most common use case of `goto` is, however, exiting from nested loops and blocks such as `for`, `foreach`, `do`, `while`, and `switch` statements.

For example,

```
var (i, j) = (0, 0);
while (++i < 10) {
    while (++j < 10) {
        if (i + j > 15) {
            goto Done;
        }
    }
}
Done: Console.WriteLine("Done");
```

26.11. The `break` Statement

The `break` statement exits the nearest enclosing `for`, `foreach`, `do`, `while`, or `switch` statement.

```
break;
```

When multiple `for`, `foreach`, `do`, `while`, or `switch` statements are nested within each other, a `break` statement applies only to the innermost statement. In order to transfer control across multiple nesting levels, `goto` statements with properly-placed target labels can be used. In certain cases, the `return` statement can also be used to exit all nested loops or `switch` statement.

26.12. The `continue` Statement

The `continue` statement starts a new iteration of the innermost enclosing `for`, `foreach`, `do`, or `while` statement.

```
continue;
```

For example,

```
int[] lucas = { 2, 1, 3, 4, 7, 11, 18, 29, 47 };
foreach (var i in lucas) {
    if (i % 2 == 0) {                         ①
        continue;
    }
    Console.WriteLine($"Odd Lucas number: {i}");
}
```

① This `if` statement skips all even numbers.

When multiple `for`, `foreach`, `do`, or `while` statements are nested within each other, a `continue` statement applies only to the innermost loop. In order to transfer control across multiple nesting levels, `goto` statements can be used.

26.13. The `return` Statement

The `return` statement can only be used inside a function invocation, and it returns control to the current caller of the function.

```
return;
return EXPRESSION;
```

A `return` statement without an expression can only be used in functions with `void` return type. If a `return` statement specifies an expression `EXPRESSION`, the this expression is evaluated first, and its value is converted to the effective return type of the function by an implicit conversion. The resulting expression becomes the result value of the function call expression.

26.14. The `lock` Statement

The `lock` statement in C# is often used, in a multithreaded programming environment, to execute a code segment while holding a mutual-exclusion (mutex) lock. In particular, a `lock` statement executes code in three steps:

- It first obtains a mutex lock for an object,
- Executes the given code, and then
- Releases the lock.

```
lock ( EXPRESSION ) STATEMENT
```

The expression `EXPRESSION` of a `lock` statement must be a value of a reference type. The statement `STATEMENT` can be a block statement including zero, one, or more other statements.

For example,

```
class SocialMedia {
    int likes = 0;
    public void thumbsUp() {
        lock (this) {
            ++this.likes;
        }
    }
    public void thumbsDown() {
        lock (this) {
            --this.likes;
        }
    }
}
```

Chapter 27. Pattern Matching

Pattern matching was first introduced in C# 7, and it has been going though iterations in every major C# release since. For example, a new " list pattern" is included in C# 11. Pattern matching is currently supported in

- The `is` expression,
- The `switch` expression, and
- The `switch` statement.

Pattern matching enables an alternative way to express control flow, e.g., compared to the more traditional `if` or `switch` - `case` statements, based on the type or "shape" of the data, or the *pattern*. If the data satisfies a pattern in a switch-case or conditional expression, the control flow selects the given branch. One can use the `when` keyword to specify additional rules to the given pattern.

A pattern belongs to one of the following dozen or so categories:

- Simple patterns:
  - Discard pattern
  - Constant pattern
  - Relational pattern
  - Var pattern
  - Type pattern
  - Declaration pattern
- Compound patterns:
  - Logical pattern
  - Parenthesized pattern

- Property pattern
- Positional pattern
- Tuple pattern
- List pattern

27.1. Discard Pattern

A discard pattern `_` can only be used with a `switch` expression, and it matches any expression. Discard patterns are usually used as the last "catch all" pattern. In an `is` expression or a `switch` statement, a var pattern, `var _`, can be used as a wild card pattern.

For example,

```
const int num = 10;
var word = num switch {
    1 => "One",
    2 => "Two",
    _ => "Too big for me",              ①
};
```

① The discard pattern `_` is used as the last catch-all pattern. In this example, the value of `word` will be `"Too big for me"`.

27.2. Constant Pattern

A constant pattern tests the value of a target expression against a constant expression such as a literal, a `const` variable, or an `enum` constant.

For example,

```
enum Season { Summer, Winter }
```

```
var season = Season.Winter;
if (season is Season.Summer) {           ①
    Console.WriteLine("It's very hot");
} else {
    Console.WriteLine("It's very cold");
}
```

① An enum value `Season.Summer` is a constant pattern.

As of C# 11, a constant string literal can match an expression of the `Span<char>` or `ReadOnlySpan<char>` types. This is officially called the "span pattern", but it is really a special case of constant patterns. For instance,

```
ReadOnlySpan<char> greeting = "hello";
var language = greeting switch {         ①
    "hola" => "Spanish",                 ②
    "salut" => "French",
    _ => "English",
};
```

① The type of the expression `greeting` is `ReadOnlySpan<char>`.

② `"hola"` and `"salut"` are constant string literals.

27.3. Relational Pattern

A relational pattern comprises a relational operator, `<`, `>`, `<=`, and `>=`, and a constant expression. The constant expression can be of an integer, floating-point, char, or `enum` type. For example,

```
const int myIQ = 200;
var judgement = myIQ switch {
    > 160 => "Smarter than Einstein",    ①
    <= 0 => @"A funny looking pattern.   ②
```

```
            But, unrealistic.",
    _ => "Just another average genius",
};
```

① This relational pattern `> 160` will match if `myIQ` is bigger than `160`. And, it does because `myIQ == 200` in this example.

② Another example of a relational pattern, `<= 0`, which will match if `myIQ` happens to be a non-negative number.

27.4. Var Pattern

A var pattern consists of the contextual keyword `var` followed by a new local variable. It matches any expression (when used without `when`), and its result is assigned to the specified local variable.

Var patterns are generally used to "capture" a value from the matched expression. The type of the declared local variable in a var pattern is the compile-time type of the expression that is matched against the pattern.

```
var rand = new Random();
var raw = rand.Next(-10, 10);
var halfAndHalf = raw switch {            ①
    < 0 => 0,                             ②
    var r => r + 1,                       ③
};
```

① The `halfAndHalf` variable ends up being assigned a random number, with a 50% chance of being `0`. Otherwise, it will be an integer between `1` and `10` with (more or less) equal probabilities.

② A relational pattern.

③ A var pattern, `var r`. This matches any number, and the value of `r` is that matched number.

Sometimes, var patterns with the `when` guards may be preferred over relational patterns, especially when you need to "capture" certain matched values. For instance, in the following example, we use a var pattern, `var r when r < 0`, instead of the relational pattern `< 0` in the previous example.

```
var rand = new Random();
var raw = rand.Next(-10, 10);
var (value, comment) = raw switch {
    var r when r < 0 => (0, $"Ignoring a negative value,
{r}"),
    var r => (r + 1, "No comment"),
};
```

27.5. Type Pattern

A type pattern checks the run-time type of a target expression. For example,

```
var array = new object[] { "hell", "o" };
if (array is string[]) {                    ①
    Console.WriteLine("I am a string array.");
} else {
    Console.WriteLine("I am not a string array.");
}
```

① The type pattern, `string[]`, matches if `array` is a type `string[]`. In this example, since `object[]` is not `string[]`, the pattern does not match.

Note that the type pattern is used to check the polymorphic types at run time (e.g., supertypes vs subtypes, interface types), and hence it is primarily useful for reference type variables.

27.6. Declaration Pattern

A declaration pattern is essentially a combination of a type pattern and a var pattern. It consists of a type followed by a local variable (similar to the local variable declaration syntax).

A declaration pattern matches if the type of the target expression is compatible with the given type at run time and, if so, the expression is cast to the given type first, and then the resulting value is assigned to the given local variable. For example,

```
object[] array = new[] { "hell", "o" };
if (array is string[] strArray) {        ①
    var upper = strArray.Select(s => s.ToUpper()).ToList();
    upper.ForEach(Console.WriteLine);
} else {
    Console.WriteLine("I am not a string array.");
}
```

① A declaration pattern, `string[] strArray`. If `array` is compatible with the `string[]`, it will match, which is the case in this particular example. If the declaration pattern matches, `array` is cast to `string[]`, and the value is finally assigned to the local variable `strArray`. This *captured* variable can be used within the `if` block in this example.

Note that a declaration pattern is semantically equivalent to a two-step operation, casting with `as` and checking if the casting has been successful. The above example code is, for instances, effectively the same as the following:

```
object[] array = new[] { "hell", "o" };
var strArray = array as string[];        ①
if (strArray is not null) {              ②
    var upper = strArray.Select(s => s.ToUpper()).ToList();
    upper.ForEach(Console.WriteLine);
} else {
    Console.WriteLine("I am not a string array.");
}
```

① The as operator is discussed earlier in the Expressions chapter.

② This is a logical pattern. These two lines serve exactly the same purpose as the declaration pattern in the above example.

27.7. Logical Pattern

You can use the `not`, `and`, and `or` pattern combinators to create compound patterns, called the logical patterns, much the same way that we can create complex Boolean expressions using the logical operators, `!`, `&&`, and `||`. In terms of operator precedence, the pattern combinators are ordered, `not`, `and`, and `or`.

27.7.1. Negation `not` pattern

The pattern `not P` matches if `P` does not match, and vice versa. The following example shows an idiomatic use of the `not null` pattern to check if an expression is non-null:

```
string? name = "Ronaldo";
if (name is not null) {                    ①
    Console.WriteLine($"Hello, {name}!");
}
```

① The null-check via `x is not null` is more or less the same as `x != null`. But, note that the `==` and `!=` operators can be overloaded, meaning that depending on the specific type of `x`, `x != null` may return a value that is not what you intend. On the other hand, `x is not null` *always* checks the nullness of `x`.

27.7.2. Conjunctive and pattern

The pattern `P1 and P2` matches an expression when both patterns, `P1` and `P2`, match the target expression. For example,

```
var degree = 37.0;
var h2o = degree switch {
    >= 100 => "hot steam",
    > 0 and < 100 => "bath water",       ①
    _ => "frozen ice",
};
```

① This logical pattern `> 0 and < 100` combines two relational patterns, `> 0` and `< 100`, through logical AND. Note that, in this particular example, `> 0 and < 100` is effectively the same as a simpler pattern `> 0`.

27.7.3. Disjunctive or pattern

The pattern `P1 or P2` matches an expression if pattern `P1` matches, or if not, `P1 or P2` matches if `P2` matches. Otherwise, the `or` pattern does not match. For instance,

```
// enum Day { Mon = 1, Tue, Fri = 5 }
// enum Mood { Happy, NotSoHappy, Miserable }

var day = Day.Fri;
var mood = day switch {
    Day.Mon or Day.Tue => Mood.Miserable,          ①
    Day.Fri or (Day) 6 or (Day) 7 => Mood.Happy, ②
    _ => Mood.NotSoHappy,                          ③
};
Console.WriteLine($"My mood = {mood}!");
```

① An `or` pattern with two constant patterns.

② The pattern `Day.Fri or (Day) 6 or (Day) 7` is equivalent to `(Day.Fri or (Day) 6) or (Day) 7` with essentially two `or` patterns. We go over the parenthesized pattern in the next section.

③ To cover all values of the `Day enum` exhaustively.

27.8. Parenthesized Pattern

Patterns can be combined in various ways to create more complex patterns. One of the simplest ways to do so is to use parentheses `()` around a pattern (e.g., so that it can be combined with other patterns). This is called a parenthesized pattern. For example, parenthesized patterns can be used to to emphasize or change the precedence in logical patterns.

```
var (rand, r) = (new Random(), 0);          ①
while ((r = rand.Next(-10, 10)) is          ②
    (>= -5) and (< 0 or >= 5)) { }          ③
Console.WriteLine($"Some weird random number = {r}");
```

① A local variable declaration with tuple deconstruction. The parentheses are used as part of tuples.

② An assignment expression enclosed in parentheses. The value of this expression is the same as that of `r` after the assignment.

③ The two pairs of parentheses used in this line are parenthesized patterns. The first one is not really needed, that is, `(r >= 5)` is the same as `r >= 5` in this context. But, the second pair is necessary due to the fact that `and` has a higher precedence over `or`.

27.9. Property Pattern

Property patterns are used to match on an object's properties or fields against nested patterns. A property pattern, using curly braces `{}`, matches a target expression when the expression is non-null and every nested pattern matches the corresponding property or field of the expression. For example,

```
var date = DateTime.Now;
if (date is { Month: 12, Day: 25 or 31 }) {     ①
    Console.WriteLine("Today is a New Year's Eve! Or, maybe
just a Christmas?");
}
```

① The property pattern, `{ Month: 12, Day: 25 or 31 }`, would match `date` if its properties `Month` and `Day` are `12` and `25` or `12` and `31`, regardless of the year or the time.

A property pattern is a recursive pattern. That is, you can use property patterns to match a property of a property of an object, etc. For instance,

```
// public record Point(int X, int Y);
// public record Circle(Point O, int R);

var circle = new Circle(new(1, 0), 10);
if (circle is { O: { X: 0 } or { Y: 0 } })      ①
{
    Console.WriteLine("Hey, I'm on the X-axis! Or, maybe am I
on the y-axis?");
}
```

Alternatively, the above property pattern could have been written slightly more succinctly using the "extended property pattern" (available since C# 10). For example,

```
var circle = new Circle(new(0, 4), 100);
if (circle is { O.X: 0 } or { O.Y: 0 })         ①
{
    Console.WriteLine("Hey, I don't know~~~");
}
```

① A pattern `{ O.X: 0 }` is equivalent to `{ O: { X: 0 }}`.

27.10. Positional Pattern

Some types have a `Deconstruct` method defined that deconstructs their properties into discrete variables. C# natively supports deconstruction of tuples. It also automatically creates a `Deconstruct` method for record types.

When a value or an object can be deconstructed, a positional pattern can be used to inspect its properties and match against the corresponding nested patterns. For instance, using the `Point` record type from the previous section,

```
// public record Point(int X, int Y);

var point = new Point(1, 0);
var affirmation = point switch {          ①
    (0, 0) => "I'm Origin",               ②
    (_, 0) => "I'm X-axis",               ③
    (> 0, _) => "I'm a Half Plane",       ④
    (>= -5 and < 5, >= 10 and < 20) =>    ⑤
        "I'm a Square!",
    _ => "I am FREE~~~",                  ⑥
};
```

① The object `point` is "deconstructible". More specifically, in can be deconstructed to the `X` and `Y` properties.

② Two constant patterns inside a positional pattern.

③ A positional pattern comprising discard and constant patterns.

④ A pattern with a relational pattern and a discard pattern.

⑤ A pattern with two logical patterns, each with two relational patterns connected with `and`.

⑥ A catch-all discard pattern.

27.11. Tuple Pattern

The tuple pattern is essentially identical to the positional pattern, but the context is slightly different. In a pattern matching expression or statement, such as `is` or `switch`, the target expression itself may be a tuple. In such a case, we can use positional patterns directly without requiring a deconstruction of the target expression first.

Using the same example from the previous section,

```
var (x, y) = point;
var affirmation = (x, y) switch {
    // ...
}
```

Or,

```
var affirmation = (point.X, point.Y) switch {
    // ...
}
```

27.12. List Pattern

The list pattern, introduced in C# 11, is similar to the positional pattern, and it lets you match an array or list against a sequence of other patterns. For example, an expression `abc is ['a', 'b', 'c']` is true when `abc` is an array or a list of 3 *char* items, `'a'`, `'b'`, and `'c'`.

Unlike in the positional pattern, it can match over arrays and lists of the same element types, but possibly with different lengths.

You can match elements using any pattern, including constant, type, property and relational patterns. In addition, the discard pattern `_` matches any single element, and the slice pattern `..` matches any sequence of zero or more elements. The slice pattern can only be used within a list pattern. Note further that no more than one slice pattern can be included in a list pattern.

Here's an example illustrating some list patterns:

```
int[] expr = { 1, 2, 3 };
var matched = expr switch {
    [1, 2, 3] => "[1, 2, 3]",                       ①
    [1, _, 3] => "[1, _, 3]",                       ②
    [1, 2, > 0] => "[1, 2, > 0]",                   ③
    [var x, 2, 3] => $"[var x, 2, 3], x: {x}",      ④
    [int, _, 3] => "[int, _, 3]",                   ⑤
    [int x, _, _] => $"[int x, _, _], x: {x}",      ⑥
    [1, .., 3] => "[1, .., 3]",                     ⑦
    [.., 3] => "[.., 3]",                           ⑧
    [_, ..] => "[_, ..]",                           ⑨
    [..] => "[..]",                                 ⑩
    _ => "_",                                       ⑪
};
Console.WriteLine($"matched = {matched}");
```

① This pattern, `[1, 2, 3]`, comprising three constant patterns will match an array or a list of 3 items, `1`, `2`, and `3`, but no others.

② This pattern, including one discard pattern `_` will match a sequence of 3 items, with the first and third ones `1` and `3`, respectively.

③ This pattern includes a relational element pattern in its third position. The first two elements need to match exactly, whereas the third element should be an integer greater than `0`.

④ Another pattern with the var pattern for its first element.

⑤ Another three element pattern.

⑥ Another three element pattern with a declaration pattern.

⑦ This pattern can match a sequence with 2 or more elements. As long as its first and last elements are `1` and `3`, respectively, it will match.

⑧ Likewise, this pattern will match any sequence with at least one element whose last element is `3`.

⑨ This pattern, `[_, ..]`, will match any sequence of any type as long as it has at least one element.

⑩ This pattern, `[..]`, matches any sequence.

⑪ The wildcard, or discard, pattern matches any expression.

27.12.1. Slice pattern

The slice pattern has two forms: `..` and `.. PATTERN`. Both will take up the remaining space after matching other elements and it will try to match the sequence in that space. In case of `..`, it matches any sequence. In case of `.. PATTERN`, it still needs to match the `PATTERN`. Here's a simple example using the second form of the slice pattern. The `Len<T>` function recursively computes the length of the given array.

```
static int Len<T>(T[] list) => list switch {
    [] => 0,
    [_, .. (T[] tail)] => 1 + Len(tail),       ①
};
```

① This is a commonly-used `[head: tail]` pattern. Since we are not interested in the actual value of the head, we use the discard pattern, and we just add `1` for the discarded head. The second element of this list pattern is a slice pattern comprising a parentheses pattern, which in turn includes a declaration pattern.

Chapter 28. Using & Disposable

The `using` statement is used to obtain a resource, execute statements using that resource, and then dispose of it at the end. There are two forms of the `using` statement.

28.1. The `using` Statement

```
using ( EXPRESSION ) {
    STATEMENTS
}
using ( TYPE VARIABLE = EXPRESSION ) {
    STATEMENTS
}
```

The `using` statement is a compound statement that includes an embedded statement. As in some previous examples, we have replaced it with a block statement because that is the most commonly used form. `STATEMENTS` represents zero or more statements. `TYPE VARIABLE = EXPRESSION` is a local variable declaration. `TYPE` can be a type, including `dynamic`, or the `var` keyword. `VARIABLE` is read-only. The type of `EXPRESSION` should be a "disposable resource", which is

- Either a class, record, or struct which has an accessible `Dispose` instance method, or
- A type that implements the `System.IDisposable` interface, which includes a single `Dispose` method.

The `Dispose` method needs to implement any cleanup code for the resource, and the code that is using the resource can explicitly call `Dispose` to indicate that it is done using the resource. Or, within the `using` statement, the compiler generates the call to `Dispose()`, which is automatically called when control leaves the `using` block.

28.2. The `using` Declaration

In an alternative form, the `using` statement can be used as a declaration, with the following syntax.

```
using TYPE VARIABLE = EXPRESSION;
// Use the resource until the end of the current block
```

This is a local variable declaration prefixed with the keyword `using`. As with the first form, `EXPRESSION` evaluates to a *disposable resource* type. The scope of the read-only local variable `VARIABLE` is the block that encloses this `using` declaration. The resource is disposed of when `VARIABLE` goes out of scope, e.g., at the end of the enclosing block.

In both forms of the `using` statement, if control leaves the current block before it reaches the end, either through an exception or other control flow actions, the `Dispose()` method is automatically called.

Here's a simple type that implements `IDisposable`:

```
record class PaperCup() : IDisposable {
    public void UseMe() {
        Console.WriteLine("Use me! I'm disposable");
    }
    public void Dispose() {
        Console.WriteLine("I'm being disposed~~");
    }
}
```

Now, we can use an object of this type in the `using` statement.

```
using var cup = new PaperCup();
cup.UseMe();
```

Chapter 29. Exception Handling

The C#'s exception framework uses the `try` - `catch` - `finally` statement to deal with any unexpected or exceptional situations. C# causes exceptions in certain circumstances during the program execution when the operation cannot be completed normally. Exceptions can also be explicitly thrown using the `throw` expression.

29.1. The `Exception` Base Class

The `System.Exception` class is the base type of all exceptions in .NET, which includes the following instance constructor:

```
public Exception(string? message, Exception? innerException);
```

- `Message` is a readonly property of the `string?` type that is used to provide a human-readable description of the reason for the exception.
- `InnerException` is a readonly property of the nullable `Exception` type. This property is used for "exception chaining". That is, when the current exception was raised in the catch block of a `try-catch` statement, its `InnerException` property is used to refer to the original exception that caused the current exception.

29.2. The `throw` Expression

C# supports both `throw` expressions and `throw` statements. The `throw` statement with an exception argument is syntactically identical to the `throw` expression used as a statement.

```
throw EXPRESSION;
```

A `throw` expression throws the value produced by evaluating `EXPRESSION`. It must denote a value of `System.Exception` or of a type that has `System.Exception` as its effective base class.

A `throw` statement with no argument can be used only in a `catch` block. This form of `throw` statement re-throws the exception that is currently being handled by that `catch` block.

```
throw;
```

29.3. The `try` - `catch` Statement

The `try` statement with `catch` clauses is used for catching exceptions that occur during execution of a block. In addition, a `finally` clause can be used to specify a block of cleanup code that is always executed whether an exception occurred or not. Here's a general structure of the `try` statement.

```
try {                                       ①
    STATEMENTS
} catch (EXCEPTION) {                       ②
    STATEMENTS
} catch (EXCEPTION NAME) {                  ③
    STATEMENTS
} catch (EXCEPTION) when (EXPR) {           ④
    STATEMENTS
} catch (EXCEPTION NAME) when (EXPR) {      ⑤
    STATEMENTS
} catch {                                   ⑥
    STATEMENTS
} finally {                                 ⑦
    STATEMENTS
}
```

① A `try` statement consists of the keyword `try` followed by a block, optional `catch` clauses, and an optional `finally` clause. Syntactically, at least one `catch` or `finally` clause is required. The statements in the `try` block can throw exceptions.

② A `catch` clause can have a few different forms. In this particular form, `catch (EXCEPTION) { STATEMENTS }`, an exception type is specified, within a pair of parentheses, after the `catch` keyword, which are followed by a block. `EXCEPTION` should be `System.Exception` or its subtype. If a thrown exception is a subtype of `EXCEPTION`, this catch clause "catches" the exception, and the corresponding block statements are executed. All other following `catch` blocks will be ignored after that `catch` clause.

③ In the exception specifier, an identifier (`NAME`) can be optionally included after the exception type, similar to the local variable declaration syntax. In such a case, the identifier, which refers to the caught exception, can be referenced in the `catch` block.

④ An exception can be caught conditionally using a filter, which consists of the contextual keyword `when` followed by a parenthesized boolean expressions (`EXPR`). In such a case, the `catch` clause can catch the exception only if `EXPR` evaluates to `true` in addition to the exception type match.

⑤ The exception variable can also by optionally specified, and this local variable can be referenced both in the `when` clause as well as in the `catch` block.

⑥ There can be at most one general `catch` clause without a particular exception type in a `try` statement. This must be the last `catch` clause before a `finally` clause, if any. This `catch` clause catches any exception type.

⑦ The statements in the `finally` block, if present, are always executed regardless of whether an exception is thrown from the `try` block.

Here are some quick examples:

```
class BallException : Exception {                        ①
    public BallException(int code, string? message) :
base(message) => Code = code;
    public int Code { get; }
}
```

① You can create an exception type by inheriting from `Exception`.

```
static int Deliberately(int color) => color switch { ①
    1 or 2 => throw new BallException(code: color, $"Color
code is {color}"),
    _ => throw new Exception("Curve ball"),
};
```

① A helper function to deliberately throw some exceptions.

```
foreach (var i in new[] { 1, 2, 3 }) {                   ①
    try {
        Deliberately(i);                                 ②
    } catch (BallException ball) when (ball.Code == 1) {
        Console.WriteLine($"ball = {ball.Message}");
    } catch (BallException ball) {
        Console.WriteLine($"ball = {ball.Code},
{ball.Message}");
    } catch (Exception ex) {
        Console.WriteLine($"Exception = {ex.Message}");
    } finally {
        Console.WriteLine("Throw again");
    }
}
```

① In each of these three cases, a slightly different exception is thrown, and they are caught by different `catch` blocks.

② Generally speaking, the `try` statement is used to guard against exception-throwing statements.

Chapter 30. Attributes

In C#, user-defined types of declarative information, called *attributes*, can be attached to program entities such as types and members, or local functions, depending on the attribute declarations. This attribute information can then be retrieved in a run-time environment.

This is similar, for example, to the way the accessibility of a method is specified via declarative access modifiers such as `public`, `protected`, and `private`, etc.

- Attributes are defined through the declaration of attribute classes, which can have positional and named parameters,
- Attributes are attached to various entities in a C# program using attribute specifications, and
- They can be retrieved at run time as attribute instances.

30.1. Attribute Classes

An attribute class is, by definition, a class inheriting from the abstract class `System.Attribute`, either directly or indirectly. Although it is not required, many attribute class names end with the suffix `Attribute` by convention. When the attributes are used, this suffix can be omitted.

For example,

```
class UselessAttribute : Attribute { }    ①
class GenericDemo<T> : Attribute { }      ②
```

① This attribute can be used with a name `Useless` or `UselessAttribute`.

② As of C# 11, we can now create and use generic attributes.

An attribute class can be decorated with other attributes, just like other regular classes. In particular, `System.AttributeUsageAttribute` can be used to describe how a given attribute class can be used. `AttributeUsage` specifies what kind of program entities which the given attribute can be used with, using a positional parameter.

For instance,

```
[AttributeUsage(AttributeTargets.Interface |
AttributeTargets.Class)]
class EmptyAttribute : Attribute {}      ①
```

① The `Empty` attribute can be used with a class or interface, but no other entities. For example,

```
[Empty] class C1 { }
[Empty] interface I1 { }
```

The `AttributeUsage` attribute also has a named parameter, `AllowMultiple`, which indicates whether the attribute can be specified more than once for a given entity. If its value is explicitly set to `true`, then the given attribute class is a *multi-use attribute class*, and it can be specified more than once on the same entity. Otherwise, that attribute class is a *single-use attribute class*.

30.1.1. Attribute inheritance

Attributes are "inherited" by default. That is, if a class inherits from a base class which is decorated by an attribute, then that attribute also applies to the inherited class.

This behavior can be changed by setting the value of `AttributeUsage`'s named parameter, `Inherited`, to `false`.

30.1.2. Reserved attributes

In addition to `AttributeUsage`, there are a few other attribute classes defined that affect the language in the `System`, `System.Diagnostics`, `System.Diagnostics.CodeAnalysis`, and `System.Runtime.CompilerServices` namespaces.

`System.Diagnostics.ConditionalAttribute`

`Conditional` is a multi-use attribute class which is used to define conditional methods and conditional attribute classes. This attribute indicates a condition by testing a conditional compilation symbol.

`System.ObsoleteAttribute`

`Obsolete` is used to mark a member as obsolete or deprecated.

`System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute`

`SetsRequiredMembers` specifies that the given constructor sets all `required` members for the current type, and callers do not need to set any `required` members themselves. (New in C# 11.)

`CallerLineNumberAttribute`, `CallerFilePathAttribute`, `CallerMemberNameAttribute`, `CallerArgumentExpressionAttribute`

These attributes from the `System.Runtime.CompilerServices` namespace are used to supply information about the calling context to optional parameters.

30.2. Attribute Parameters

Attribute classes can have *positional parameters* and *named parameters*.

- Each public instance constructor for an attribute class defines a valid sequence of positional parameters for that attribute class.

- Each non-static public read-write field and property for an attribute class defines a named parameter for the attribute class.

30.2.1. Attribute parameter types

The types of positional and named parameters for an attribute class are limited to the following:

- One of the following types: `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `sbyte`, `short`, `string`, `uint`, `ulong`, `ushort`.
- The type `object`.
- The type `System.Type`.
- Enum types.
- Single-dimensional arrays of any of the above types.

30.3. Attribute Specification

Attributes can be specified

- At global scope, to specify attributes on the containing assembly or module, and
- Locally, for
  - Type declarations,
  - Member declarations for `interface`, `class`, `record`, `struct`, and `enum`,
  - Accessor declarations,
  - Event accessor declarations,
  - Elements of formal parameter lists,
  - Elements of type parameter lists, and
  - Local function declarations (New in C# 11).

30.3.1. Attribute sections

Attributes are specified in *attribute sections*. An attribute section consists of a pair of square brackets `[]`, which encloses a comma-separated list of one or more attributes. Multiple attribute sections can be applied to a program entity. The order of the attribute sections, as well as the order of the attributes in a given attribute section, is not significant. For example, the attribute specifications for the following four properties are all equivalent:

```
class Balloon {
    [Up, Down] public int Nav1 { get; }
    [Down, Up] public int Nav2 { get; }
    [Up][Down] public int Nav3 { get; }
    [Down][Up] public int Nav4 { get; }
}
```

30.3.2. Attribute targets

By default, the target of an attribute is the entity that follows the attribute specification. But, one can explicitly specify an attribute target using the following syntax:

```
[ TARGET : COMMA_SEPARATED_LIST_OF_ATTRIBUTES ]
```

The valid values for `TARGET` are `assembly`, `module`, `field`, `event`, `method`, `param`, `property`, `return`, and `type`. Since C# 11, attributes can be specified on the backing fields of the auto-properties using the `field` target. Here's an example.

```
[Up] class Balloon {
    [field: Down] public int Nav1 { get; }
}
```

A. How to Use This Book

> Tell me and I forget. Teach me and I remember. Involve me and I learn.
>
> — Benjamin Franklin

The books in this "Mini Reference" series are written for a wide audience. It means that some readers will find this particular book "too easy" and some readers will find this book "too difficult", depending on their prior experience related to programming. That’s quite all right. Different readers will get different things out of this book. At the end of the day, learning is a skill, which we all can learn to get better at. Here are some quick pointers in case you need some advice.

First of all, books like this are bound to have some errors, and some typos. We go through multiple revisions, and every time we do that there is a finite chance to introduce new errors. We know that some people have strong opinions on this, but you should get over it. Even after spending millions of dollars, a rocket launch can go wrong. All non-trivial software have some amount of bugs.

Although it’s a cliche, there are two kinds of people in this world. Some see a "glass half full". Some see a "glass half empty". *This book has a lot to offer.* As a general note, we encourage the readers to view the world as "half full" rather than to focus too much on negative things. *Despite* some (small) possible errors, and formatting issues, you will get *a lot* out of this book if you have the right attitude.

There is this book called *Algorithms to Live By*, which came out several years ago, and it became an instant best seller. There are now many similar books, copycats, published since then. The book is written for "laypeople", and illustrate how computer science concepts like specific algorithms can be useful in everyday life.

Inspired by this, we have some concrete suggestions on how to best read this book. This is *one* suggestion which you can take into account while using this book. As stated, ultimately, whatever works for you is the best way for you.

Most of the readers reading this book should be familiar with some basic algorithm concepts. When you do a graph search, there are two major ways to traverse all the nodes in a graph. One is called the "depth first search", and the other is called the "breadth first search". At the risk of oversimplifying, when you read a tutorial style book, you go through the book from beginning to end. Note that the book content is generally organized in a tree structure. There are chapters, and each chapter includes sections, and so forth. Reading a book sequentially often corresponds to the *depth first traversal*.

On the other hand, for reference-style books like this one, which are written to cover broad and wide range of topics, and which have many interdependencies among the topics, it is often best to adopt the *breadth first traversal*.

This advice should be especially useful to new-comers to the language. The core concepts of any (non-trivial) programming language are all interconnected. That’s the way it is. When you read an earlier part of the book, which may depend on the concepts explained later in the book, you can either ignore the things you don’t understand and move on, or you can flip through the book to go back and forth. It’s up to you. One thing you don’t want to do is to get stuck in one place, and be frustrated and feel resentful (toward the book).

The best way to read books like this one is through "multiple passes", again using a programming jargon. The first time, you only try to get the high-level concepts. At each iteration, you try to get more and more details. It is really up to you, and only you can tell, as to how many passes would be required to get much of what this book has to offer.

Again, *good luck!*

Index

About the Author

Harry Yoon has been programming for over three decades. He has used over 20 different programming languages in his academic and professional career. His experience spans broad areas from scientific programming and machine learning to enterprise software and Web and mobile app development.

He *occasionally* hangs out on social media:

- Instagram: @codeandtips [https://www.instagram.com/codeandtips/]
- TikTok: @codeandtips [https://tiktok.com/@codeandtips]
- Twitter: @codeandtips [https://twitter.com/codeandtips]
- YouTube: @codeandtips [https://www.youtube.com/@codeandtips]
- Reddit: r/codeandtips [https://www.reddit.com/r/codeandtips/]

Other C# Books by the Author

- The Art of C# - Basics: Introduction to Programming in Modern C# - Beginner to Intermediate [https://www.amazon.com/dp/B08X2SCG2Y]

About the Series

We are creating a number of books under the series title, *A Hitchhiker's Guide to the Modern Programming Languages*. We cover essential syntax of the 12 select languages in 100 pages or so, Go, C#, Python, Typescript, Rust, C++, Java, Julia, Javascript, Haskell, Scala, and Lua.

These are all very interesting and widely used languages that can teach you different ways of programming, and more importantly, different ways of thinking.

All Books in the Series

- Go Mini Reference [https://www.amazon.com/dp/B09V5QXTCC/]
- Modern C# Mini Reference [https://www.amazon.com/dp/B0B57PXLFC/]
- Python Mini Reference [https://www.amazon.com/dp/B0B2QJD6P8/]
- Typescript Mini Reference [https://www.amazon.com/dp/B0B54537JK/]
- Rust Mini Reference [https://www.amazon.com/dp/B09Y74PH2B/]
- C++20 Mini Reference [https://www.amazon.com/dp/B0B5YLXLB3/]
- Modern Java Mini Reference [https://www.amazon.com/dp/B0B75PCHW2/]
- Julia Mini Reference [https://www.amazon.com/dp/B0B6PZ2BCJ/]
- Javascript Mini Reference [https://www.amazon.com/dp/B0B75RZLRB/]
- Haskell Mini Reference [https://www.amazon.com/dp/B09X8PLG9P/]
- Scala 3 Mini Reference [https://www.amazon.com/dp/B0B95Y6584/]
- Lua Mini Reference [https://www.amazon.com/dp/B09V95T452/]

Community Support

We are building a website for programmers, from beginners to more experienced. It covers various coding-related topics from algorithms to machine learning, and from design patterns to cybersecurity, and more. You can also find some sample code in the GitLab repositories.

- www.codeandtips.com
- gitlab.com/codeandtips

Mailing List

Please join our mailing list, join@codingbookspress.com, to receive coding tips and other news from **Coding Books Press**, including free, or discounted, book promotions. If we find any significant errors in the book, then we will send you an updated version of the book (in PDF). Advance review copies will be made available to select members on the list before new books are published.

Request for Feedback

If you find any errors or typos, or if any part of the book is not very clear to you, or if you have any general suggestions or comments regarding the book, then please let us know. Although we cannot answer all the questions and emails, we will try our best to address the issues that are brought to our attention.

- feedback@codingbookspress.com

Please note that creating and publishing quality books takes a great deal of time and effort, and we really appreciate the readers' feedback.

Revision 1.1.3, 2023-04-04