The Art of C# - Basics

Introduction to Programming in Modern C# - Beginner to Intermediate

Harry Yoon

Version 2.0.1, 2023-03-15

Copyright

The Art of C# - Basics:

Introduction to Programming in Modern C#

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: July 2021

Harry Yoon
San Diego, California

Preface

Learn C# programming for fun!

C# is a very interesting programming language. It is indeed *fun* to program in the "modern C#". That is, if you know how to *really program in C#.*

The Art of C# - Basics: Introduction to Programming in Modern C# - Beginner to Intermediate is an attempt to introduce the modern C# to the masses, so to speak. This book is written for a broad audience. Whether you are new to programming, or whether you have been programming for some time using other languages, you will have to learn C# as a new modern language, as of 2022. As of version C# 10.0.

The Art of C# - Basics starts from the absolute basics and moves on to more advanced topics. The book provides a comprehensive introduction to "the modern C#". It is not a language reference. It is not a grammar book. The book does not merely teach the language syntax.

This book teaches the fundamentals of programming in Modern C# and its idiomatic uses. Although it is an introductory book, you will gain sufficient knowledge and perspective that you can venture into a journey of real programming in C# on your own.

The Art of C# - Basics: Introduction to Programming in Modern C# teaches the important "concepts". But, not necessarily all the details. This book teaches you "what", but not necessarily "how" exactly in every situation. The goal of the book is to give you a real taste of what the modern C# is, and teach you how to get started with professional C# programming. This book will not be the only resource in your effort to learn and master programming in C#. But it will be an invaluable part.

Knowledge is familiarity.

Throughout this book, we will introduce certain terms and concepts without precisely defining them first and then we will elaborate on them later in the book. You do not have to learn, understand, or memorize, everything on your first encounter. This book teaches programming in C# *through repetitions*. Readers are encouraged to read the book through, from beginning to end, or at least the first two parts, without interruption. The details are not as important as you might think.

There are a lot of programming "technicians". I hope that the readers of this book strive to become "artists". The artists in programming.

Good luck!

Revision 2.0.1, 2023-03-15

Table of Contents

Introduction

> Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.
>
> — Linus Torvalds

C# is a rather complex language. There is no other way to put it. C# includes a lot of "features". Once you master the basics, however, it will be one of *the easiest languages to use.* It will make you more productive like no other languages. You can achieve the same things with less code in less time in many cases.

Are You a Beginner?

If you are new to programming, and thinking about learning C#, then welcome! It will be a hack of a fun ride.

The Art of C# - Basics: Introduction to Programming in Modern C# starts from the basics and progresses to more advanced topics. Although it is written to be a beginner's book, the caveat is that if you haven't had any experience in programming then you can expect some bumpy ride. The book tries to explain (most) everything from scratch (as far as the basic programming goes), but it will take some effort and commitment on your part to get benefits from this book.

The book does not tell you how to install Visual Studio, for example, or how to create new Visual Studio projects or solutions. It does not teach you what keyboard shortcuts to use for certain tasks.

The main focus of this book is to teach you the fundamentals of programming. *Real* programming in C#. As with any other skills, the foundation is the most important thing. This book will teach you the best practices in C# programming.

Are You an Experienced Developer New to C#?

If you are a seasoned programmer with experiences in other programming languages, and want to try out C#, then welcome! You picked the right book as your first guide to the wonderful world of C#.

The Art of C# - Basics covers the essential elements of the C# language. Regardless of your background, the book will provide a gentle introduction to the "idiomatic" style of programming in Modern C#.

If you learn any unique features of C# that you haven’t seen in other programming languages, after reading this book, and you are intrigued enough to decide to learn more of C#, then the book will have served its purpose.

Have You Used C# Before?

Last but not least, if you have used C# before, and if you are now interested in using it again, then welcome back! The modern C# is not your grandfather’s C# as the saying goes. C# has been around for over two decades now, and it has gone through many small and big changes. The current major version is C# 10.0, as of this writing.

The Art of C# - Basics provides the most up-to-date introduction to the C# programming language. This book will primarily focus on *the current C#* as is, mostly ignoring the historical baggages, if you will. This book will present C#, as one programming language, not as a patchwork of the C# 1.0 features and the C# 2.0 features … and the C# 10.0 features.

> If you have some programming experience, either in C# or in other languages, the book might seem too verbose or too slow-

> moving. If you are familiar with the basic concepts of programming, then you can probably skip some early parts of the book without losing too much continuity. There are a fair amount of repetitions in the book, *by design*.

Reading Before "Premature Writing"

Learning a programming language is not much different from learning a foreign language. You learn from examples, primarily by listening and reading.

The same principle applies to learning programming languages. You learn from examples, primarily by reading the well-written code.

Most of the examples used in this book are, although small and elementary, all inspired by the real world use cases, to varying degrees. The readers are encouraged to "read" the code samples first, not just the text in the book, *before* diving into the main part of each lesson. Much of the code may not make much sense at the first reading. But, that is how we learn a new language. That is how we learn a new skill.

The Topics Covered in This Book

The Art of C# - Basics: Introduction to Programming in Modern C# - Beginner to Intermediate is organized into a series of small *lessons*. Each lesson teaches basic concepts of programming, and programming in C# in particular, by going through carefully-designed sample code.

The lessons gradually progress from the basic topics to the more advanced subjects. You can advance at your own pace. If you are just starting out, then you can take your time. If you have some programming experience, then you can skip, or skim over, some of the earlier lessons, as stated.

The book covers a lot of topics. But it is not meant to be a language reference of the C# programming language. This is not an academic textbook. In fact, there are many topics that we leave out for the sake of brevity.

The Art of C# - Basics covers the following topics, among other things:

- The basic structure of a C# program.
- Top level statements.
- Command line interface. Basic input/output processing.
- Basic constructs of the C# language such as expressions and statements.
- Primitive types, strings, arrays, and tuples.
- Custom types, in particular, enums, structs, classes, and records.
- Value types vs reference types.
- Interfaces. Delegates.
- Collections. Enumerators. LINQ. Lambda expressions.
- Error handling.
- Object oriented programming. Inheritance. Polymorphism.
- Generics. Type variance. Type constraints.
- Pattern matching. Switch expressions.
- Async programming.
- File-scoped namespace.
- Global using. Implicit using.

Basic Concepts of C# Programming

Programs written in different programming languages are processed and run differently. For example, a program in C is compiled into a machine-readable binary, specific to a given machine architecture. Java programs are compiled first into the byte code for the JVM (Java virtual machine), which is then compiled into the machine code at run time.

The C# programs are processed rather similarly to those of Java. They are compiled into the Microsoft IL code (intermediate language code) first, and the IL code is then run on the common language runtime (CLR). But, a modern C# program can also be compiled/packaged into a machine executable.

The C# programs can be run in a number of different environments or "runtimes". .NET is one of the most common runtimes that support the C# programming language.

In this book, we will mostly use .NET 6. .NET has many different flavors and versions. .NET 5 and 6 are, at least in theory, the next generation of .NET that is the (eventual) successor of both the .NET Framework (Windows only) and the .NET Core (cross-platform) as well as Mono/Xamarin.

.NET is a *cross-platform runtime*. You can run .NET 5 and 6 on Linux and Mac as well as on Windows. Therefore, C# effectively has the cross-platform support. Any programs that you write in C# can run on different operating systems.

We will mostly use the command line tool, *dotnet*, in this book to manage and build the C# programs. *dotnet* is also a cross-platform tool. (If you would like to install the .NET SDK on your computer, which includes the dotnet CLI, then there is a quick note at the end of Lesson 1.)

The C# compiler can now handle non-nullable references and nullable references differently. It was first introduced in version 8.0, under the name "the nullable references" (or "the nullable context"), and it is an "opt-in" feature. Although it appears to be a small change, it is indeed one of the most significant updates in the C#'s' 20+ year history.

We will use this feature throughout this book. It is more than likely that it is here to stay. The nullable context will be the future of C#. One thing to note is that enabling the nullable context feature can potentially make your programs incompatible with the older way of compilation. There is no turning back. ☺

Book Organization

The lessons in this book are organized into four parts: First Steps, Moving Forward, Having Fun. and Final Projects.

This division is somewhat arbitrary, but the first part, Part I: First Steps, mostly focuses on general programming. If you are coming from a different programming language background, especially languages like C++ or Java, then you will feel comfortable with the lessons in this part. One thing to note is that we introduce "generics" rather early in the book, unlike many "introductory" beginner’s books. Generics is an essential part of the modern C# programming language.

The second part, Part II: Moving Forward, mainly focuses on the object oriented style programming in C#. In some object oriented programming (OOP) languages, there is only one or two constructs, such as `class` or `struct`, to create a custom type. In C#, there are several. In particular, C# has `class`, `struct`, `record`, and `interface`, among others.

In Part II, we discuss the fundamental concepts of object oriented programming, and the C#'s support for the OOP. In addition, this part introduces some features in C# that support the functional programming stylea. We will also go over some basic examples of Web programming in C# as well in this part.

In Part III: Having Fun, we will cover a few independent topics, mostly to help the readers internalize the important programming concepts in C#, like object-oriented and functional programming styles.

Finally, in Part IV: Final Projects, we will work on a few hands-on projects. We will design and implement a program to solve Rubik’s Cube, among other things. The readers are encouraged to tackle all the projects in this part. They are excellent projects for beginning programmers as well as more experienced developers.

Notes on C# 10.0

This book was completely revised after the release of C# 10.0, in the late 2021 and early 2022. Although the C# changes were relatively minor, there were still a few parts in the book that required some substantial updates.

Most notably, C# 10.0 introduced another way to create a custom type, namely, `record struct`. This is essentially a combination of the C# 9.0 `record` (which can now be alternatively declared as `record class`) and the `struct`.

Another major change is that C# now allows overriding the parameterless default constructor of the `struct` type. This can have some far-reaching implications depending on how you have been using structs.

Furthermore, C# 10.0 also includes the features known as the "global using", "implicit usings", and "file-scoped namespace" declarations. They are mostly "cosmetic", but they can change the general look of any C# code (along with the C# 9.0 top-level statement feature). And hence they are really significant changes.

We will discuss these features throughout this revised edition.

Final Remarks

Regarding the sample programs, one thing to note is that we do not include comments in the code. Comments will mostly add clutter in books like this. The content of the book serves as code comments. And, more. In practice, writing good comments, and documentations, is a very important skill to learn.

All exercises are "optional" in that the lessons in the book mostly focus on the "code reading", and not writing. The exercises may also require knowledge on some subjects that we do not thoroughly cover in this book.

As stated, C# may not be the easiest language to learn. It has a lot of "features". There are many different ways to use C# for the same task. It will take time and effort to become really proficient in the modern C#. Hope you find this book helpful in your journey into the programming world, in C#.

Let’s get started.

Part I: First Steps

The beginning is the most important part of the work. -Plato

The C# programs run on the .NET runtime, among others. A "runtime" can include both the build time and run time support, including the "standard libraries". We will mostly use the .NET runtime version 6 in this book.

We will use the C# version 10.0, which are the default C# language version on .NET 6. The .NET is a cross-platform runtime, meaning that it is not restricted only to the Windows platform. You can run any modern C# programs on .NET, using the .NET libraries.

The word "program" can mean, depending on the context, a binary program, e.g., an executable on a particular computer architecture, or a source program, which is typically a collection of source code files, (Or, even a part of the complete program.)

A C# binary program can be either an executable or a library. In the .NET world, they are called the "assemblies". An assembly can be either an executable (e.g., typically with the file name extension `.exe` on Windows) or a dynamically linked library (e.g., typically with the extension `.dll` on Windows).

C# is a class-based programming language. A `class` plays an essential role in C#. The `class`, and its cousins, allow the programers to create custom types, which are the core components of any C# programs.

In this first part, we will primarily focus on the programming styles using the built-in or library-supplied types. We will start defining and using our own custom types from the second part.

> The code samples in this book are meant to be *read*. You do not have to type them on the computer to get the "hands-on"

> experience. Just to be clear, however, you will need to make conscious efforts for "reading" (that involves "thinking" ☺). Otherwise, the benefits will be minimal.

Chapter 1. Hello World 1

1.1. Agenda - The Simplest C# Program, `Console.WriteLine`

Of course, we will *have to* start with a "hello world" program. ☺

1.2. Code Reading - Hello World 1

This is the simplest C# program.

hello-world-1/Program.cs

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

Throughout this book, the label of a source code snippet indicates that the example code is taken from a certain file in a certain folder (on the author’s computer). The particular source file names, and the file paths, are largely irrelevant to the execution of a C# program.

If you are an absolute beginner, then a C# "source file" is just a text file, with a ".cs" extension, which includes the source code. Writing a program means writing one or more of these source files.

1.2.1. Explanation

The program comprises a single source file, "Program.cs". The file contains a single line of code. It cannot be simpler than this.

You can quickly run the program as follows using the dotnet CLI tool (assuming that you have it installed on your computer), from the same directory where the source file is in:

```
dotnet run
```

It produces the following output:

```
Hello World!!!
```

Yes, as promised, it is indeed a hello world program. The program prints out a text "Hello World!!!" to the console and it terminates.

Throughout this book, we will primarily use the command line interface (CLI). If you are not familiar with any "terminal program", then this might be a good time to learn some basics specific to your platform (e.g., Linux, Mac, or Windows).

The book focuses on the fundamentals of programming, and the CLI is more than sufficient for illustrating the basic, and even advanced, programming concepts. Incidentally, there is a quick note at the end of this lesson regarding how to set up a " development environment".

This book is *for reading*. The author does all the work. 😀 He even includes all the relevant outputs from all (or, most of) the sample programs in this book. The readers can just sit back and "read". ☺

1.2.2. Grammar

A C# program primarily contains the "custom type definitions", using constructs like `struct` or `class`, as we will learn throughout this book.

This hello world program is an exception, however. It merely includes one "statement". A statement is an instruction to the computer as to what needs to be done. The statements written in a program like this, instead of being included in other constructs like classes or methods, are called the "top level statements" in C# (C# 9.0 and later).

An executable C# program can have at most one source file that contains the top level statements. Furthermore, the top level statements, if present, must be in the beginning part of the source file, before any custom type definitions.

The top level statements can also include static method definitions, as we will illustrate in the next lessons. The top level statement feature (new as of 9.0) makes the main part of a program look a bit simpler by eliminating some boilerplate code.

In this example, the single-line top-level statement is a "method call", `Console.WriteLine("Hello World!!!")`. (The pair of parentheses (`(` and `)`) is used for the "method invocation". In this case, `WriteLine` is the method.)

The statement appears to include two components separated by a dot (`.`), namely `Console` and `WriteLine("Hello World!!!")`. In fact, this statement is a shorthand for

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

This includes three components. The first part, `System`, is called a "namespace". The System namespace is from the standard .NET library, and it includes many different types and what not. The `Console` class happens to be one of them. The namespaces are used to reduce the chances of name collisions across different C# programs (e.g., executables and libraries).

In order be able to use the `Console` class, its name should be qualified with the namespace, as in the example: `System.Console`. Alternatively, one can use the keyword `using` to "introduce" (all) the names from the given namespace into the program, or more precisely to the specific source code file. For example,

```
using System;

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

In this example, by using the `using` declaration, we can refer to the `Console` class without having to fully qualify its name. Note, however, that certain system namespaces are treated as special, as of C# 10.0, including the `System` namespace. They need not be explicitly imported using the `using` declaration. As in the original example code, we can just omit the `using System` declaration. This is known as the "implicit using", "implicit global using", or "implicit namespace import", in C# 10.0.

This is an "opt-in" feature, and we will discuss this further a bit later.

`Console` happens to be what we call a "static class". It does not really define a "type". It merely includes a number of "static methods" and other static variables and constants. In this example, we use the static method `WriteLine()` defined in the `Console` class to print out a text to the console (hence the name). Note again the dot syntax, connecting the names of the static class and its static method.

In languages like C# and Java, one cannot define a top-level *function*, which is an essential part of most other programming languages like C/C++, Go, Python, and Javascript. As a workaround, if you will, one can use the "static methods" in liu of the top level functions. In fact, the static methods in C# and Java can be viewed as the functions in other programming languages. It's mostly their syntax that are different.

The syntax for using the names from a static class can be simplified by using the `using static` declaration. For instance,

```
using static System.Console;

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

By introducing the name `System.Console` into the source code file using the `using static` declaration, now we can refer to the static method `WriteLine()` of `Console` without having to qualify its name.

Note that we are using the fully qualified name of `System.Console`, not just the class name, in the `using static` statement, as required by the C# grammar.

> This feature is officially called the "static using". The author prefers to call it the `using static` declaration because that is how he uses it when he programs. The terminologies are important, but only to the extent that they help clarify the

> underlying concepts and/or as long as they facilitate easier communications.
>
> Throughout this book, we will just use the terms that make sense (although they may not be the "official" terms). At the end of the day, it is the program that speaks, not the names that we attach to the parts of the program.

Now, as for the static method `WriteLine()` of the `Console` class defined in the `System` namespace, it takes one argument of the `string` type (in this example), and it prints out the given argument(s) to the terminal. `"Hello World!!!"` (with a pair of double quotes) is a "string literal". It represents a specific (constant) value of the type `string`.

That's all there is to it. If you build and run the program, then it prints out the given string (`Hello World!!!` in this case, without the double quotes) to the console, and since there is nothing more to do, the program terminates.

One thing to note is that many statements in C# end with a semicolon (`;`). It is required in many types of statements. (Also, a "block" can terminate a statement instead of the semicolon.)

At the risk of stating the obvious, the semicolon is required at the end of a statement in a C# program. It is not the end of a line where we need to use a semicolon. (Just to reiterate, some types of statements in C# require (explicit or implicit) "blocks" and those types of statements do not end with semicolons.)

> If you are new to programming, then you might be already overwhelmed. Even for this supposedly "simplest program" with one line of code, a lot seem to be going on. No worries. You don't have to "understand" everything, or even *anything*. ☺ As stated, knowledge is familiarity. With the exposure to more examples over time, you will gradually become more comfortable with

> what the "namespace" is, what the "using" declaration is, what the "static method" is, and so forth.
>
> It should also be noted that we do not provide the "dictionary definitions" of these concepts in this book. You will learn them through examples. In the proper contexts.
>
> When you see an apple, you know it's an apple. You do not need to know the precise dictionary definition of the apple to recognize one.

1.3. Code Reading - A New C# Project

One can use the IDEs like Microsoft's Visual Studio or JetBrain's Rider for developing C# programs. Although it is not strictly necessary, most IDEs, and the `dotnet` CLI tool, use the "projects" to manage the C# programs, and their library dependencies, etc.

We will use the `dotnet new` command to create a project in this lesson. For example,

```
dotnet new console
```

This command (in an empty folder) creates a "console app" project. .NET supports a few different languages such as F# and Visual Basic, but C# is the default, and the above command, without any options, creates a C# project.

Here's an example project file (in XML), created by `dotnet` version 6.0.100:

hello-universe-0/hello-universe-0.csproj

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

```

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <RootNamespace>hello_universe_0</RootNamespace>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

We ran the *dotnet new* command in the directory named `hello-universe-0`, and it used the name as a project name by default. Running this command also created a boilerplate C# code file, named "Program.cs":

hello-universe-0/Program.cs

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

This example uses C# 10.0 and .NET 6. The prior versions, for example, C# 9.0/.NET 5, used different templates, e.g., without using the top-level statements and the implicit `System` namespace using. We will briefly discuss this in the next few lessons.

1.3.1. Explanation

The C# project file is used for various purposes. As we will see later, one of the most important roles of a C# project is to help manage the (third-party) library dependencies.

Another use of the projects is to set the global settings across the different C# source code files and the build outputs, etc. For example, for this "hello-universe-0" project, the output is set to "Exe", indicating that the project is to produce an executable program, or an "application".

```
<OutputType>Exe</OutputType>
```

Its target is set to "net6.0" (because we used the version 6 of `dotnet`). That is, we are targeting the .NET 6 runtime.

```
<TargetFramework>net6.0</TargetFramework>
```

`net6.0` is a cross-platform target that provides, and uses, the common denominator of all the APIs. For any previous version of the .NET, for example for 5.0, you can set the target framework to the appropriate version, e.g., `net5.0`. If we set the target to, for instance, `net6.0-windows`, then we could additionally use certain Windows APIs (like for GUI programming), but our program may not run on other platforms like Linux or Mac.

As mentioned, the "implicit global using" setting is an *opt-in* feature. This is most likely due to the backward compatibility concerns. You can enable it in the project files as follows.

```
<ImplicitUsings>enable</ImplicitUsings>
```

This is enabled by default if you scaffold your app using the *dotnet* CLI or the Visual Studio project wizard. If you are starting a new project, then there is no reason not to use this feature. Virtually every programming language has some similar support except, maybe, C and C++. It reduces a lot of redundant boilerplate code.

If you set the `ImplicitUsings` flag to `enable`, then the following system namespaces are automatically, and "globally", imported.

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

We will use some of these namespaces later in the book. This implicit using feature *implicitly uses* (pun intended ☺) another new C# 10.0 feature called the "global using". The word "global" means, in this context, project-wide. The normal `import` statement has effects only in a given source code file. Prior to C# 10.0, you had to import the same namespaces in every source file which used those namespaces. Now, you can import them using the `global import` statements in one file, and it works across all the files in the project. We will discuss this further later in the book.

One other thing to note in the generated project file is the `Nullable` flag. This enables the "nullable reference type" feature.

```
<Nullable>enable</Nullable>
```

This was first introduced in C# 8 and 9, and it is becoming the standard way of programming in C#. This is also an opt-in feature for the backward compatibility reasons, but it is highly recommended you always enable it for new projects. (And, it is enabled by default if you use *dotnet new* or other tools to create a new C# project.) More on this later.

Now the generated *Program.cs* file contains a simple placeholder code, which is the same "hello world" one-liner program that we used earlier. Although you cannot see (obviously, if you think about it icon:grin{}), it uses the *implicit global using* feature. It also uses the "top level statements".

If we used an older version of *dotnet* CLI, or other tools, we would have ended up with the following boilerplate code.

```
using System;

namespace hello_universe_0
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Note the contrast. This program does exactly the same as the one-liner that we just saw, and no more. And yet, it is *astronomically* more complicated. 😊 Let's take a quick look.

The program starts with the usual `using System` declaration, which can be omitted for C# 10.0 or later. Then, there is a namespace block:

```
namespace hello_universe_0
{
    // ...
}
```

The namespace block is enclosed within a pair of curly braces (`{` and `}`). Anything that is included within this namespace block belongs to this namespace. Starting from C# 10.0, however, we can also declare a namespace for the entire source file. For example,

```
namespace hello_universe_0;
// ...
```

In this case, anything that is declared/defined in the file, after this `namespace` declaration, will belong to that namespace. This is called the "file-scoped namespace" in C# 10.0.

Note that the namespace name (generated by `dotnet new`) is based on the folder name, `hello-universe-0` in this example, but they are slightly different. The dash or minus sign (`-`) cannot be used in an identifier, or name, in C# programs. Hence it is converted to the underscore symbol (`_`).

Different programming languages have different rules, but for most C-style languages including C#, a valid identifier comprises one or more alphanumeric characters. Special symbols are not allowed in a name, but the underscore `_` is an exception. An identifier cannot start with a number character/digit (since the numeric literals, or numbers, start with a number character).

Another thing to note is that it is more typical to use the "PascalCase" for namespace names in C# as well as for the custom type names, etc. The PascalCase naming convention, aka the upper "CamelCase", uses a concatenation of one or more words as a name in which all words are capitalized. As in "PascalCase".

On the other hand, the lower camelCase naming convention follows the same rule as that of the PascalCase, but it uses a starting lowercase letter. As in "camelCase". In C#, the camelCase names are typically used for the variable names, among other

> things.

Within this (optional) namespace, the `dotnet new console` command created a simple `class` named "Program".

```
class Program
{
    // ...
}
```

The definition of the class `Program` is also enclosed within a pair of curly braces (`{` and `}`). C# is a *block-scoped language.*

This class `Program` happens to include a single static method named `Main()`:

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

Note the syntax. It starts with the keyword `static` indicating that it is declaring a static method. A static method is just like a "function" in other programming languages like C/C++, Go, Javascript, or Python. It has little to do with the "object oriented programming" (despite the fact that we use "class").

The next keyword `void` indicates that the method does not return any value. Then comes the name of the function, `Main` in this case, followed by a pair of parentheses (`"("` and `")"`). Within the parentheses, we include the parameter list. When we call this method, we will need to provide proper values for these parameters. For this particular method, the list includes a single parameter of the string array type (`string[]`).

The parameter name is mostly not significant (to the compiler), but it serves as a documentation and hence it is recommended to pick the meaningful names when you create your own methods.

> We use the terms "parameters" and "arguments" interchangeably in this book. Some people prefer to use the terms like these with (subtly) different meanings. But, we do not. The contexts in which these terms are used are more important than the precise (English) words used and their ad-hoc meanings (in programming). Another such example is the terms "functions" vs "methods", for instance. Some people may swear by their "precise" definitions. We do not care. ☺

The `Main()` static method is special. It serves as an entry point when starting the program. We do not explicitly call the `Main()` method. The .NET runtime, or the operating system, will call it with the command line arguments, if any, as we will discuss in the next lesson. The parameter name is, by convention, `args`, although you can change it to any arbitrary valid variable name.

In this generated code, the `args` argument is never used (although the intention is to let the programmer add the real implementation in the placeholder method body), and hence we can omit it (for now).

```
static void Main()
{
    Console.WriteLine("Hello World!");
}
```

The `Main()` methods in C# should follow one of the few predefined "method signatures". Both `static void Main(string[] args)` and `static void Main()` are valid for the `Main()` method. We will discuss other valid signatures later in the book.

If we run this generated program,

```
dotnet run
```

It prints out, yes, "Hello World!" to the console.

```
Hello World!
```

1.4. Code Reading - Hello World 1 Redux

In the first example of this book, we primarily used the (relatively new) top-level statement syntax. Let’s see how they look in more traditional implementations.

> We will discuss the object-oriented programming style, in depth, in Part II.

The one-liner program "hello-world-1", `System.Console.WriteLine("Hello World!!!");` using the top level statements, or `Console.WriteLine("Hello World!!!");` (relying on the implicit `System` namespace import), is equivalent to the following:

hello-universe-1/Program.cs

```
class Program {
    static void Main() => Console.WriteLine("Hello World!!!");
}
```

As stated, the namespace blocks are optional, and we generally omit them in the " main programs". Namespaces are mostly needed for libraries. This program includes one `class` named "Main" as in the generated boilerplate program.

1.4.1. Explanation

We do not always need to use `dotnet new` when creating a new project. We can, for example, just copy the project XML file from an existing project and use it as a template.

Here's a C# project file for this project:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

This is essentially the same as the one scaffolded by the *dotnet* tool.

We can run the program as follows:

```
dotnet run
```

The output is the same as before:

```
Hello World!!!
```

1.4.2. Grammar

If you are new to the "modern C#", then the program may look a bit strange.

This example uses a (new-ish) syntax called the "expression-bodied method". When a method body includes one statement (or, more precisely, one expression or a "return" statement), we can do away with the method body block and just use an expression in liu of the method body block.

It uses a syntax similar to a "Lambda expression", as we will see later in the book, using the "fat arrow" symbol (`⇒`). This sample code is equivalent to the following:

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

Note that a method call (e.g., `WriteLine("…")`) is an expression, which can also be used as a statement. In this alternative, and more traditional, example, `Console.WriteLine("Hello World!!!");` is a statement. Note the semicolon at the end.

In the expression-bodied method, on the other hand, `Console.WriteLine("Hello World!!!")` is an expression. The semicolon in this case belongs to the entire statement, `static void Main() ⇒ Console.WriteLine("Hello World!!!");`.

C# now supports a number of different "expression body" syntax. The right hand side of the fat arrow operator is an expression in all these cases. If the only statement in a method happens to be a return statement, then the return value (an expression) can be used in the expression body without the `return` keyword.

Using this new syntax is generally preferred (when applicable) because they tend to reduce the code clutter and they typically make the code easier to read (although it may require some getting-used-to for some people).

Summary

In this lesson, we learned the simplest way to write an output to the console, namely, using `Console.WriteLine()` methods.

We also learned the main C# program structure, either using the "top-level statements" or using the more traditional `Main()` static method. The top-level statements are merely a syntactic sugar for the implicitly defined `Main()` method in an implicitly defined class.

An executable C# code, either on the .NET runtime or as a native binary, needs to have one, and only one, `Main()` method, either explicitly or implicitly, which is the entry point to the program.

We also briefly looked at a way to define a method using the "expression body", if applicable, instead of using the (more traditional) "block body".

1.5. Exercises

1. Write a program that prints out a text "!Hola mundo!". You can refer to the sample code of "hello-world-1". But do not copy and paste.

> All exercises in this book are optional. In fact, it is best to skip the exercises in your first reading, especially if you are new to programming.

Author’s Note

Development Environment Setup

We will not discuss how to install C# tools in this book, or how to use IDEs, etc. That is a "trivial" (albeit important) task, which you can figure out using various other resources (e.g., Web search). Having a C# development environment on your computer, or even owning a computer ☺, is not a prerequisite to reading this book.

If you haven't installed the C# tools, and if you would like to do so, then there are a few different options. You can use Visual Studio. Or, you can install the .NET SDK, which includes the C# development tools. You can find a quick instruction on the Microsoft website: Install .NET on Windows, Linux, and macOS [https://docs.microsoft.com/en-us/dotnet/core/install/]. The examples in this book primarily use the `dotnet` CLI tool, for concreteness.

If you use VSCode as a code editor (Visual Studio Code, not to be confused with Visual Studio), and if you feel really "adventurous" ☺, then you can even try the "dotnet interactive notebook" through a VSCode extension (currently, beta). Here's the link: .NET Interactive Notebooks [https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.dotnet-interactive-vscode]. This tool allows you to try out simple C# programs without you having to install the C# compiler tools on your machine.

(Well, if you have installed VSCode on your computer, then the chances are that you have already installed the C# extension, C# by Microsoft [https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp]. Tools like "Interactive Notebooks" can become really useful when we can use them without requiring a "dev machine".)

There are many different types of resources that can help you learn programming in C#. ***The Art of C# - Basics: Introduction to Programming in Modern C#*** is designed/created to be *read*. This book is not meant to be used as a stand-alone be-all and end-all book.

Tip - Learn "Modern C#"

> This book includes a number of "tip" sections. They are more like informal essays, and they are not part of the main lessons of the book. These tips can be skipped, if you like.

C++, the grandfather of C#, was created in the mid 1980’s, which was supposed to be an "improvement" over the C programming language, as indicated by the `++` increment operator in the name. C had been around since the early 1970’s.

The second edition of the book, The C++ Programming Language, by the language creator Bjarne Stroustrup, came out in 1991, and C++ started to get more and more popular throughout the 1990’s. The "Object Oriented Programming" style was all the craze. Everybody wanted to get on the OOP bandwagon.

During that time, however, there were no compilers that implemented all the features that were described in the book. Things were in flux. Compiler tool vendors like GNU and Microsoft were slow to catch up. No reliable implementation of the standard template library (STL), for instance, was available until the mid/late 1990’s.

Finally, the then-de-facto-standard C++ was officially standardized as C++98 in 1998. It was mostly what the second edition of The C++ Programming Language specified (in 1990/1991). The language was stable since then, and it appeared that it was going to remain that way, just like C had been.

Then came Java. Then came C#.

These were the languages directly based on, or influenced by, the C++ programming language. And, they were created as the "OOP" languages from the ground up, unlike C++, which was an extension of C. These were "modern" languages in various respects, compared to the languages like C and C++.

Throughout 2010’s, and beyond, C++ went through dramatic changes, C++11, C++14, C++17, and C++20. We call them the "modern C++" in contrast with C++98. These changes have been presumably influenced by the advent of these new programming languages as well as by the advances in the programming language theories and the progresses in the compiler technologies. We expect C++23 soon, and there will be new versions every 3 years.

This kind of rapid changes in the programming languages was unimaginable in the early days. Just imagine the world where the English grammar changes every 3 years. Clearly, the programming languages are different from the natural languages. But how much? In what way?

C# was created around 2000, and it has been going through hyper-rapid changes ever since.

C# was originally created based on Java. C# 1.0 was pretty much indistinguishable from Java. Now, C# 10.0 is completely different. The idiomatic way of programming in C# is very different from that in Java (although Java has been evolving as well with some similar updates).

There is no "modern C#". It is just a play of words. C# is constantly changing. There will be a new major release *every year*.

If you cannot keep up, then unfortunately C# may not be for you. You do not have to retrain yourself, or refresh your programming style or C# knowledge, every year. But, with some reasonable lag, you will still have to keep up in order to be able to keep using C# productively in the long run. (Note that you do not create software by yourself in isolation. You use other people’s libraries, and so forth.)

If you are just starting to learn C#, then there is no reason not to start with the current, most up-to-date, version.

In fact, it is absolutely recommended to start learning the most recent language version. Even if you do not constantly keep up with the ongoing changes in C#, you will have at least a few years of buffer. (And, learning takes time.)

You do not want to learn the older versions (much of which may be still useful at this point). The older features, and styles, may become (gradually) outdated or obsolete soon. The exception is when you’ll have to maintain the old codebase, etc. But even then learning the most up-to-date C# should be your first step, not the other way around.

> When we say C# 10.0, we mean the C# with all the features from 1.0 to 10.0 (minus anything deprecated or otherwise made obsolete). The C# 10.0 is, however, not a mere superset of C# 1.0, or C# 5.0. The way you program in C# 10.0 can be, and will be in many cases, rather different from the way you program in, say, C# 6.0.
>
> That is why the "chronological" learning approach, starting from all the 1.0 features, and moving on to newer 2.0 features, etc., is not the best method when learning languages like C# which have been rapidly changing over a span of two decades.

If you are looking for resources to learn C#, e.g., books like this, then make sure that they cover the newest features of C#. Check their publication dates, for example. Not just the features. Make sure that they use the code samples, and coding styles, that are "modern" and current.

At the core, every programming language (based on C) has common features. They have conditional statements. They provide loops such as "for" or "while". They support arithmetic operations like additions and multiplications. And so forth. These features predate even C, and they exist in virtually all "imperative" programming languages.

At the core, every version of C# has certain common features. C# is C#.

However, that is not the proper way to use a programming language. If you are using every (C-style) language in the same way, then you are clearly using them wrong.

The language features exist to help make programmers more productive.

If you are using only C# 6.0 or older features, for instance, then you are clearly missing out. The best practices and programming styles in C# 10.0 are *fundamentally* different from those of C# 6.0. Every new release introduces gradual updates, but over time they amount to significant changes.

> Even if you are absolutely new to programming, or to the C# programming language, do not waste your time with the old C#. Old books. Old resources. Old C# programming styles and idioms. Learn the "new C#". **Modern C#.**

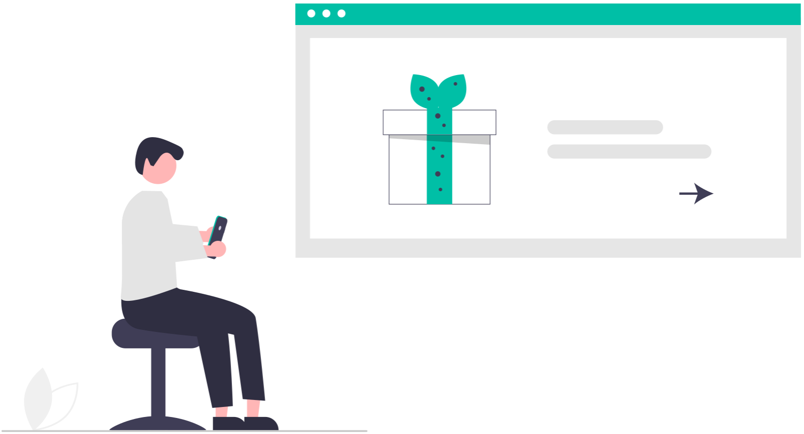
We are *not* going to cover which new features were introduced in what specific versions of C# in this book. Instead, we will mostly focus on the "idiomatic" way of programming in C# as of the most recent release (although it is a moving target), using all the features available to us programmers from 1.0 to 10.0.

One thing to note is that, throughout the 20+ year evolution of C#, the trend has been to put more emphasis on the functional programming styles. Among the many C language descendants, C# is probably the only language that enables a reasonable degree of functional programming styles.

> And, as we will see throughout this book, these new changes, which support functional programming, are some of the most important features of the "modern C#".

Chapter 2. Hello World 2

2.1. Agenda - Variables, Constants, Strings, String Interpolation



2.2. Code Reading - Hello World 2

Here's a slightly more complicated version of the "hello world" program.

hello-world-2/Program.cs

```
const string name = "Joe";
var greeting = $"Hello {name}!!!";

Console.WriteLine(greeting);
```

This source code file has the same name "Program.cs" as before, but it is stored in a different directory, "hello-world-2". The file name "Program.cs" has no special significance. A source file that includes the "main part" of the program is typically named "Program.cs".

ℹ The line numbers, printed on the left-hand side, are not part of the program. They are included for easy reference in this book.

2.2.1. Explanation

This example program includes a few more lines of code. You can run the program as before:

```
dotnet run
```

It produces the following output:

```
Hello Joe!!!
```

2.2.2. Grammar

As in the earlier examples, this source code file, the only file in the program, includes the "top-level statements" (and, nothing else). When you build and run the program, the top-level statements will be executed, more or less from top to bottom.

This program also includes the `Console.WriteLine()` statement, and we rely on the implicit namespace inclusion. That is, the `using System` declaration is implied, as specified in the project XML file.

The first line of the program defines a "string constant" named `name`.

```
const string name = "Joe";
```

Note the syntax. This statement starts with the C# keyword `const`, followed by a type name, `string` in this case, and the name of the constant, `name` in this case. The left-hand side of this statement declares a new constant `name` with a type `string`. It is then initialized with a value "Joe" (a string literal), as indicated by the equal sign (`=`), which is known as the "assignment operator" in C#.

A `string` is essentially a sequence of characters. The C#'s built-in string type corresponds to the `System.String` class on .NET.

The values of the "constants" are determined at compile time, and they do not change during the execution of the program. In C#, the variables of a type `string` and other "numeric types" like `int` and `float` can be declared as `const`.

On the other hand, the value of a "variable" declared with `var` can change. For example,

```
var greeting = $"Hello {name}!!!";
```

This statement, starting with the keyword `var`, declares a new variable named `greeting`, and it initializes the variable with the value given on the right hand side of the assignment operator, at run time.

The expression on the right hand side is what is called the "$-string interpolation". It starts with `$"` and ends with `"`. A variable, or an expression in general, enclosed in a pair of curly braces (`{}`) is replaced by its value. In this example, the value of `name` is `"Joe"`. And hence the value of the right hand side string interpolation expression is `"Hello Joe!!!"`.

A quick note on the terminology: A `const` constant can be viewed

> as a special kind of a "variable". Hence we do not always distinguish constants and variables, unless specifically needed.

C# also supports an operation called the "string concatenation". For instance, the above statement can be written as follows:

```
var greeting = "Hello " + name + "!!!";
```

Note that we use the operator `+` here, which is normally used to add numbers, to "concatenate" the string values. The complier knows what to do based on the expressions/values used in the `+` operation.

There are other ways to accomplish the same thing (e.g., using `string.Format()` method, as we will see shortly). In fact, there are many different ways, as alluded in the introduction, to do the same things in C#.

In the examples like this, we will generally prefer using the string interpolation expressions, which are easier to write and easier to read. (Note that it requires some mental effort to deduce what the result of a string concatenation would be whereas the result appears more explicit when we use a string interpolation.) In later lessons, we will show the usages of the `StringBuilder` class, which is useful, for performance reasons, when we end up doing a lot of string concatenations.

Instead of using a string literal as an argument to `Console.WriteLine()` as in the previous example, we build an output string in this case by defining a string constant and a string variable separately (for illustration purposes). Its value is stored in a variable `greeting`. Then the variable `greeting` is used to call the `Console.WriteLine()` static method as an argument.

```
Console.WriteLine(greeting);
```

The net effect is the same, and we get the same result as before when we run this program.

2.3. Code Reading - Hello World 2 Redux

The program "hello-world-2", using the top level statements, is more or less equivalent to the following:

hello-universe-2/Program.cs

```
class Program {
    const string name = "Joe";
    static void Main() => Console.WriteLine($"Hello {name}!!!");
}
```

Again, we omit the `using System` declaration although we use the `Console` class from `System` in this example because `System` is one of the automatically imported system namespaces, as long as we opt in to the feature.

2.3.1. Explanation

If you run the program with `dotnet run`, then we get the same output as before.

```
Hello Joe!!!
```

2.3.2. Grammar

These two programs, "hello-world-2" and "hello-universe-2", have some differences, but they are insignificant in this case. In fact, the hello-world-2 program and the following code would have been compiled to more or less the same IL code:

```
class Program {
    static void Main() {
        const string name = "Joe";
        var greeting = $"Hello {name}!!!";
        Console.WriteLine(greeting);
    }
}
```

Besides the fact that this version is using a temporary local variable `greeting`, the main difference is the location of the const definition `name`. In this version, the " scope" of the name `name` is limited to the body of the `Main()` method.

On the other hand, in the example code of this lesson hello-universe-2, the `name`'s scope is the entire class. If we had another method, say, after `Main()`, then we could have used the const `name` within that method as well. We will discuss the scopes in more detail throughout this book.

Summary

In this lesson, we learned the important concepts of variables and constants in C#. A variable holds a value in a C# program, which can change during the execution of the program. The value of a constant, on the other hand, does not change.

Variables, and constants, are associated with types. The type of a variable/constant is determined at build time, and it remains the same while the program is running. C# is a statically typed programming language.

2.4. Exercises

1. Write a program which keeps your name as a constant variable, stores a greeting in your native language (e.g., corresponding to "hello" in English) as

another variable, and prints out a text that combines the greeting with your name as a single string. You can refer to the sample code of "hello-world-2", but do not copy and paste.

Author’s Note

Who is This Book for?

Learning programming is hard. Learning a new language is hard. It can be frustrating. It can be discouraging. …

This book is written primarily for the beginners who have dabbled with other programming languages like Python or Javascript, or even the languages like Java or C++, but could not make a real progress in programming.

C# is a much more interesting language. Much more fun to learn and use. The language designs of other modern languages like Swift and Kotlin are largely influenced by the modern C# (although they may not admit it). As stated, C# is probably one of the most fun languages to learn programming with, and to use, as your primary programming language.

This book is also written with more experienced programmers in mind who want to learn *the modern C# programming language,* and its new programming styles.

There are a lot of resources, but none really explains what the modern C# really is. Its root from C, C++, and Java is mostly irrelevant at this point. If you want to really use C#, then you really have to learn it as a *new* language. Learning just the C# syntax will not do. You’ll need deeper understanding.

Hope you can find some interesting ideas in this book that will help you become a better C# programmer, and a better programmer in general.

We mentioned the IDEs a few times in this lesson. As stated, however, we do not use the IDEs in any of the examples in this book. In fact, the author does NOT recommend the use of *any IDEs* for the beginning programmers.

It may sound counterintuitive. But, the times are changing. It used to be the case that an IDE was a must when you started learning programming, or new programming languages.

The modern "trend" is that the simplicity is preferred. If you are just starting, or if you are a "hobby programmer", then you do not need the "full power" of the IDEs like Visual Studio or Rider.

As a matter of fact, you will benefit more, learn better, by using the simple command line tools like `dotnet` instead of the IDEs, which can obscure certain essential aspects of programming (in the name of the "convenience").

You will still need to use a good "programmer's editor". Notepad will not do. ☺ There are many choices like Sublime Text, Atom, etc. The author primarily uses VSCode [https://code.visualstudio.com/].

Tip - How to Use This Book

The Art of C# - Basics: Introduction to Programming in Modern C# is written for a broad audience, from the absolute beginners to the more seasoned developers with experience in other programming languages who want to get a quick taste of C#.

This first part, `First Steps`, in particular, is primarily for the beginners who are new to programming, or at least new to programming in C#. Depending on your experience, you may find some of the lessons too easy or trivial. Or, too slow or boring. You can skip, or skim through, certain portions of the book.

Each lesson starts with a sample code, in the "code reading" or "code review" sections. Some lessons include multiple such sections. Learning a new programming language is not much different from learning a foreign language. We primarily emphasize the "reading comprehension" skills in this book.

Through gradual exposure to many sample programs, from simple to more complex, you will become familiar with the C# language, and its idiomatic styles, without even touching a keyboard.

Here’s a good way to use this book.

For each lesson, read the sample program(s) first. If you understand what the overall program does, and know what each part of the program means syntactically, then you can skip the lesson. Go to the next lesson.

If you are new to programming, however, some parts of the book might be rather difficult, or even more or less incomprehensible. That is perfectly all right. Read the book through. (Or, at least the parts I and II.)

Then, come back to the beginning, and read the book again. This time, test your knowledge by reading the sample code first. Do you understand what the program does? If so, then you can skip this particular lesson and go to the next one. If not, that’s all right. Read the lesson again.

You do not have to understand *everything* in the lesson. As indicated, the book covers a lot of "advanced concepts", which may be considered TMI (too much information) for the beginning programmers. *Your goal should be understanding the sample code.* Everything else is a bonus.

Eventually, you will end up feeling like you understand all the sample code in this book (regardless of where you started in the first place). Then you will have learned everything you need to learn from this book.

A time to learn a new programming language. (Just kidding. 😀)

Chapter 3. Hello World 3

3.1. Agenda - Arrays, Command Line Arguments, `if` Statements

We will learn how to read the "command line arguments" in a C# program in this lesson, while continuing with the "hello world" theme.

3.2. Code Reading - Hello World 3

Here’s another variation of the Hello World program.

hello-world-3/Program.cs

```
var name = "you";
if (args.Length > 0) {
    name = string.Join(" ", args);
}

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

3.2.1. Explanation

This third Hello-World program takes an (optional) argument (although it is not obvious from reading the program). If you run the program in the usual way,

```
dotnet run
```

It produces the following output:

```
Hello you!
```

If you run the program as follows, with an extra text "Joe" in the command line,

```
dotnet run Joe
```

It produces a different output:

```
Hello Joe!
```

3.2.2. Grammar

This example program introduces a few new constructs of the C# programming language.

The keywords `if` and `else` are used for conditional statements. `if` is followed by a Boolean expression (i.e., an expression that evaluates to a `bool` value, `true` or `false`) and a "block" (from the opening bracket `{` to the closing bracket `}`).

If the Boolean or logical expression (`args.Length > 0` in this case) evaluates to `true`, then the statements in this `if` block are executed, if any. An optional `else` can be used. The statements in the `else` block are executed if the value of the Boolean expression is `false`.

For instance, in the following example,

```
if (false) {
    Console.WriteLine("I am false");
} else {
    Console.WriteLine("I am true");
}
```

It will print out `I am true`, from the else block, ignoring the statement from the `if` block. In fact, the compiler knows this (because the Boolean expression happens to be a compile time constant `false`), and it will issue a warning about the "unreachable code" since the `if` branch (in this example) will never be executed.

All operating systems, or runtimes, allow passing some kind of arguments to a starting program. These are generally known as the "command line arguments". The C programming language started using a convention where the command line arguments are passed in to the "main() function" as the function’s arguments.

The exact function signature is not important, but the C’s `main()` function accepts the command line arguments (say, multiple arguments separated by spaces) as a list of values, or an "array". Most of the C’s descendant languages follow this convention, with minor variations. C# follows more or less the same convention. The command line arguments are passed in as an argument to the `Main()` method (of the "main class"). We will discuss this in more detail in the next lesson.

For the example program using the top level statements, the command line arguments are available as an implicitly defined variable named `args`. The variable `args` is of type `string[]`, that is, an "array of strings".

> In the C convention, the first element of the command line argument array, `argv`, is the name of the program. C#, however, does not follow that convention. The `args` array contains only the command line arguments explicitly provided by the user. If none is provided, then it is an "empty array".

An "array" is a built-in type in C#. It is a collection type, so to speak. A variable of an array type stores a sequence of (zero or more) values of a given "element type" (e.g., `string` in this example) in a consecutive space in memory.

The values of its elements, which are variables themselves, can be accessed using the "index notation". For example, one can access the third element of an array `arr` as `arr[2]`, assuming that `arr` has more than 2 elements. Note that the index is "zero-based". That is, the first element of `arr` is `arr[0]`.

An array is a "reference type", as we will discuss further throughout this book.

One thing to note is that an array has a "readonly property" called `Length`. One can read the length of an array variable (i.e., the number of elements in the array) using the "dot notation". For example,

```
var length = args.Length;
Console.WriteLine($"The arr's length is {length}.");
```

The variable `name` is initialized with a value `"you"` (a string literal) in line 1 of the example code. The `if` statement next checks if the predefined array `args` has any elements (as made available to the given program by the .NET runtime or the operating system), and if so, then it updates the value of the string variable `name`.

Note that even though we use an "implicit variable declaration" using the keyword `var`, the type of `name` is `string` because it is initialized with a string value. We could have more explicitly declared `name` as a string variable as follows:

```
string name = "you";
```

In general, the `var` declaration suffices when a local variable is declared with an initial value, and it is generally preferred. On the other hand, if a variable is declared without an initial value, then `var` cannot be used. For instance, you can do this:

```
string name;
```

But, not with `var`.

> Note, however, that uninitialized local variables cannot be accessed/read until they are explicitly assigned any definite values.

Going back to the sample code, if there are any command line arguments (line 2), then we combine the command line arguments `args` into a single string, using the static method `string.Join()` (line 3).

A "static method" in C# is comparable to a function in other C-style languages, as stated earlier. Although it is defined in a class (`System.String` in this example, which is C#'s `string` type), the static method behaves in more or less the same way as functions.

The method `string.Join()` concatenates all elements of a given array (the second argument) into a single string, separated by a given "char" or `string` (the first argument). The first argument is a string literal with one character, a space (`" "`), in this case. Note that `string.Join()` has many "overloaded implementations" (as we will discuss further in later lessons), and we could have called it as follows, for instance:

```
name = string.Join(' ', args);
```

In this case, the first argument `' '` is a literal of `char` type, a space character in particular. `char` is a built-in type in C# representing the Unicode characters. More on this later.

The end result is the same regardless of whether we use `" "` or `' '` as a separator. The resulting greeting text, using the string interpolation, is printed to the console, as in the previous example.

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

> If you are new to programming, there are two kinds of methods (or functions) in the programming languages like C#. There are methods that return a value. And, there are methods that do not return anything.
>
> `System.String.Join()` happens to be a method that returns a value (a single value). It combines the elements of the second array argument with the string/char of the first argument, and it *returns* the concatenated string. In this example, the returned string value is assigned to the variable `name`.
>
> `System.Console.WriteLine()`, however, does not return any value. Outputting the argument string to the console is merely a "side effect" of calling this method. Since this method does not return any value, the only reason for calling this method is for the side effect. Just to be clear, the methods that return a value can also have side effects. Because of this, in general, the methods/functions in the imperative programming languages like

> C# are not "real functions" (in the mathematical sense).

C# is a compiled language. In particular, the `dotnet build` command on .NET (version 5 or 6, or later) generates an executable for the given machine architecture (e.g., of the machine where the build is done), as well as an IL assembly for the .NET runtime.

> The `dotnet run` command runs `dotnet build` first and then runs the generated executable. Commands like `dotnet run` and `dotnet build` are primarily for development. If you want to build an executable/assembly that can be deployed, then you use the `dotnet publish` command. Refer to the official `dotnet` CLI doc for more information. Or, you can use the `dotnet --help` command on your terminal to get a quick information.

Let's try running the `dotnet build` command:

```
dotnet build
```

Running this command, when successful, generates an executable program, or a binary. If you do *ls -l bin/Debug/net6.0*, then you can see an output like this:

```
total 180
drwxrwxr-x 3 harry harry   4096 Nov 30 13:39 ./
drwxrwxr-x 4 harry harry   4096 Aug 12 12:41 ../
-rwxr-xr-x 1 harry harry 142840 Aug 12 12:41 hello-world*
-rw-rw-r-- 1 harry harry    403 Aug 12 12:41 hello-world.deps.json
-rw-rw-r-- 1 harry harry   4608 Aug 12 12:41 hello-world.dll
-rw-rw-r-- 1 harry harry  10464 Aug 12 12:41 hello-world.pdb
-rw-rw-r-- 1 harry harry    158 Aug 12 12:41 hello-
world.runtimeconfig.json
```

```
drwxrwxr-x 2 harry harry   4096 Nov 30 13:39 ref/
```

This particular output is taken from a Unix/Linux shell (in particular, BASH), but the explanations given here, and throughout this book, are not specific to a particular platform.

As you can see, there is an executable program named *hello-world,* which was generated by the `dotnet build` command. (We can ignore all other files. We will not discuss them in this book.) If you are not familiar with Unix shells, the star at the end of the file name indicates that the file is executable (as well as the `x` characters in the file attributes).

The name of the program *hello-world* is taken, by default, from the project name "hello-world.csproj".

If you run the program as follows (e.g., in the BASH shell):

```
bin/Debug/net6.0/hello-world
```

It produces the same output as that from *dotnet run*:

```
Hello you!
```

Now, if you pass a command line argument, say, "Joe":

```
bin/Debug/net6.0/hello-world Joe
```

It produces the same output as that from *dotnet run Joe*:

```
Hello Joe!
```

What happens if we run this program with more than one arguments? Say, how about this?

```
bin/Debug/net5.0/hello-world Joe and Jill
```

We will get the following output:

```
Hello Joe and Jill!
```

When the program runs in this case, the runtime sets the implicit `args` variable to a three element array, "Joe", "and", and "Jill", as explained above. The program then combines these three elements into a single string using `string.Join()`, which results in "Joe and Jill". The output is therefore `Hello Joe and Jill!`.

What will happen if we run the program as follows: `bin/Debug/net5.0/hello-world    Joe    and    Jill`? (Note the multiple spaces between the command and the arguments as well as between the arguments.) We will leave it to the readers.

The hello-world program of this section accepts arguments. The `dotnet` CLI command also accepts optional special arguments knowns as the "options" or "flags". In some cases, the command line tool like `dotnet` may not be able to tell which are their own arguments and which are the arguments that need to be passed to the target program.

The Unix command line tool convention uses a separator string "--" (two dashes) to delineate the two. For example, we could have

> invoked our program with *dotnet run—Joe* instead of *dotnet run Joe.*

3.3. Code Reading - Hello World 3 Redux

The following code, without using the top level statements, is equivalent to the previous program "hello-world-3",

hello-universe-3/Program.cs

```
class Program {
    static void Main(string[] args) {
        var name = "you";
        if (args.Length > 0) {
            name = string.Join(" ", args);
        }
        Console.WriteLine($"Hello {name}!");
    }
}
```

3.3.1. Explanation

The behavior is exactly the same as before. If you run the program without any command line arguments, *dotnet run*, then it produces the following output:

```
Hello you!
```

If you supply the command line arguments, for example, *dotnet run—Joe and Jill*, then it produces the following output:

```
Hello Joe and Jill!
```

3.3.2. Grammar

The *top-level statements* are simply a "syntactic sugar". All top-level statements are effectively a part of an (implicit) `Main()` method in an (implicitly defined) `class`.

These two programs have no difference. As far as the compiler is concerned they are identical. They will (most likely) generate exactly the same IL code, to be run on the .NET runtime. They will (most likely) generate exactly the same executable for a given machine architecture.

The difference is, for us programmers, the code using top-level statements is just a bit easier to write, and easier to read. The only reason why we needed to use the placeholder Program class and the static Main() method was because that was required by the (pre-9.0) C# grammar.

> Note that the use of `class` in this particular example has little to do with the "object oriented programming". We simply could not write a C# program without using a `class` or something comparable. The top-level statements is the new addition to C#, which can help remove certain boilerplate code like this. As stated, C# 10.0 now includes a few more improvements in that direction.

Now that we can use this syntactic shortcut (since C# 9.0), this way of writing the main program is generally preferred. Most of the time, we do not even care if it is a mere syntactic sugar. We will just use them as if the top level statements are truly allowed in C# just like all the other programming languages like C/C++, Rust, Javascript, Python, Go, or Ruby. (Or, you name it. ☺)

(The only exception is really Java, among the commonly used programming languages, on which C# was originally based.)

We can also define and use any top-level static methods as if they are part of the implicitly defined class which includes the `Main()` method. As stated, however, we can use the top level statements in only one source code file in a program (that is, in a project). Furthermore, the top-level statements, including the top-level static methods, are limited to the beginning part of that source file (as of C# 9.0 and 10.0). Once you define a class or struct, etc. in a source file, we cannot use the top-level statements any more beyond that point.

Despite these limitations, the top-level statements are rather useful, as indicated before. And they are here to stay. We will use the top-level statements throughout this book.

The only time that we will need to think about the fact that these are not "true" top-level statements is when we need to deal with the command line arguments, etc. For instance, in the earlier example of hello-world-3, we used a "magic variable" `args`, which was nowhere explicitly declared in the program. The variable `args` is in fact an argument to the implicitly defined `Main()` method, as we can see from the hello-universe-3 example.

Summary

When we run a C# console program, we can optionally pass in one or more arguments to the starting program. We learned, in this lesson, how to read those "command line arguments" in a program.

When the .NET runtime, or the operating system, starts a C# program, it calls the `Main` method of the program as an entry point. If any command line arguments are provided, then the `args` argument of the `Main` method is set to these values, as the `string[]` type. When the top-level statements are used as an entry point to a C# program, the command line arguments are available in the implicitly defined `args` variable.

When the program is called with no arguments, this `args` argument/variable is an empty string array.

The array is a builtin type in C#, and in many other programming languages, which stores an (ordered) collection of variables. The variables, or the elements, of an array are stored in the consecutive memory space. The array type is generally the most efficient collection data structure (although it may not be the most flexible one).

One of the most powerful aspects of programming is that a single program can behave differently at run time, e.g., based on the user input or other conditions. The `if` statement is used to execute different pieces of the code depending on the runtime conditions. We will learn the `if` statement in more detail later in the book.

3.4. Exercises

1. Write a program that takes the user's name as a command line argument and prints out a text "!Hola (user's name)!". For example, given a user name "Juan", it needs to print out "!Hola Juan!". You can refer to the sample code of "hello-world-3" if you don't remember how to process the command line arguments, but do not copy and paste.

Author's Note

Who is This Book Not for?

Learning new things, or new skills, requires patience, and perseverance. And, most of all, some level of "intellectual curiosity".

If you are only interested in getting some quick knowledge on the C# programming language, then this book may not be for you.

Learning programming can be fun. Programming can be fun. And, useful. Not just as a career choice. But as a pastime, as a hobby. Just as an endeavor to satisfy your intellectual curiosities. And, to express your "creativity". *Yes, programming is a creative process.* ☺

As you might have noticed after a couple of lessons, this book takes a rather unique approach than the vast majority of other programming language books. *The Art of C# - Basics* is a book for reading. It is designed to be read just like novels, or short stories, are to be read. Or, more particularly, the book is written based on the "listen and repeat" paradigm commonly found in foreign language guide books.

This book is written to be *read.* It may not be such an outrageous concept that a book is for reading. But most technical books are written as "study materials". Most programming language books, in particular, follow more or less the same patterns. Grammars. Variables, `for` loops, `if` statements, and so on and on. These are not the books for "reading". You do not "read" dictionaries or legal code books.

On the other hand, ***The Art of C# - Basics: Introduction to Programming in Modern C# - Beginner to Intermediate*** is like the "short stories" books for learning foreign languages. The code samples are to be read. In fact, the entire book is written in such a way that you can just "read". You learn by reading, just like you read a story.

If you are looking for some quick knowledge, then this book may not be most suitable for you.

Tip - Nullable Context

A reference variable, or 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`.

(If you don't know what a "reference" is, you can ignore this section. We will discuss this throughout this book.)

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

One possible solution is to check if a variable is `null` before using the variable. In practice, however, this solution is not always feasible, especially for large scale software projects. Checking "null-ness" of every reference variable every time we use it will add substantial overhead in terms of development time and runtime cost. It will also add not-so-insignificant amount of clutter to the source code.

C# now has a compile time support (since 8.0) to address this problem. C#, and other languages like Typescript, are "waging war against the null pointer exceptions". The goal is to completely eradicate the NPEs. ☺

This new feature of C# is called the "nullable context". It is by default disabled. A missing nullable context specification means that its value is `disable`. This means that all reference variables are nullable, as they always have been.

> There are other possible values for the `nullable context` flag. But, we will not discuss them in this book. We will primarily, in fact exclusively, use the `enable` value.

Now, if you enable this feature, interesting things happen. We will discuss how to enable it shortly, but for now let's see what enabling the nullable context means.

In any code that is within the enabled nullable context, the traditional (nullable) reference variables become non-nullable. Setting `null` values to these variables will cause the compile time errors. (That is, you cannot even build the program. The compiler won't let you. Note that you cannot have runtime errors from your program unless you can build and run the program first. ☺)

In addition, one can use a new kind of reference types within the enabled nullable context. They are called the "nullable reference types". The naming is obviously confusing since all reference types are nullable in the default/traditional C# as well as in all other similar languages that support the pointer/reference types.

But, note that, within the enabled nullable context in C#, the (normal) reference types are non-nullable. Hence, it seems appropriate to use this name for this "new" kind of reference types. The names make sense only in a particular context.

All nullable reference variables (or, the variables of nullable reference types) should be checked for nullability before using them. Otherwise, the compiler will throw the compile errors or warnings (unless the compiler knows for sure that they cannot be null, e.g., through static analysis).

In theory, by doing this, we can eliminate all potential (run-time) null reference exceptions. Variables of non-nullable reference types cannot be null, and hence there is no possibilities of NPEs. For nullable reference variables, you are forced to check the nullability and do the right thing based on their null check. Otherwise, the compiler will stop you. Hence, again, you cannot have NPEs.

In practice, however, even this solution is not perfect. First of all, the compiler support is not perfect. There are a number of different scenarios where the compiler cannot help you much. Refer to the C# official doc if you are interested. But, more importantly, it is not always easy to tell if a certain variable is guaranteed to be non null. Programs have many interconnected components, one component calls/uses another, etc., and they evolve over time, often maintained by different developers or even different teams.

Another complexity comes from the way the C# grammar was written in the first place. It was originally assumed that all reference variables were potentially nullable.

For example, one will have to use certain "nullable attributes" when calling a function with a reference type argument, sometimes *before and after*. That is well beyond the scope of this book, however.

When in doubt, you should fall back to the old behavior. Assume that a reference variable can be null unless you are absolutely sure that it cannot be null. Clearly, using nullable reference variables for *all* reference variables (within the enabled nullable context) beats the purpose. Some efforts will need to be made to reduce the number of nullable reference variables.

Despite all these issues, we will use the enabled nullable context in this book, across all the examples. As stated, this could be the way we program in C#, in the not-so-distant future.

The best approach is to start using the enabled nullable context but use primarily nullable reference variables just as you have been, and to gradually decrease the use of the nullable reference variables, over time, as just stated.

> The nullable context is relevant only to the reference variables, in certain situations. Even though we use the enabled nullable context in all the examples in this book, it may not always be relevant, or even apparent. And, in many cases, it is not really visible because we cannot see what is not there. The sample code in this book mostly lack the (rather common, in fact ubiquitous) null check `if` statements.

You can enable the nullable context using the C# compile time directives. To enable, put this is in your source code file:

```
#nullable enable
```

To disable, use this:

```
#nullable disable
```

These directives are in effect until the end of the source file, or until the next `#nullable` directive is used, if any. You can also restore the previous context value using the following directive:

```
#nullable restore
```

We will not discuss C#'s compile directives in this book other than `#nullable`.

In theory, the nullable context can be enabled or disabled (or, set to other options) on a line-by-line basis. In practice, however, that kind of uses will be limited since variables are block-scoped in C#.

If you use the *dotnet* command line tool, or Visual Studio, then you can also globally enable or disable the nullable context per "project". For example, in a project XML file, you can set the "Nullable" property.

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net6.0</TargetFramework>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

In fact, if you use the new `dotnet` CLI version 6.0 or later, then all scaffolded C# projects will have the nullable context enabled by default. The compile time directives can be used to overwrite the default project-level settings.

We will discuss the details of the nullable and non-nullable reference types, in the enabled nullable context, throughout this book.

> ℹ The nullable context is purely a compile time support. Regardless of the specific context used, the C# program will end up with the same IL code (that is, if the C# compiler lets you compile it). The .NET 5/6 runtime does not know anything about the nullable context, and as far as it is concerned, all reference variables are nullable.

Chapter 4. Hello World 4

4.1. Agenda - Input Handling, String Methods

We will next look into simple input handling in a C# program.

4.2. Code Reading - Hello World 4

The following program is a bit more complicated, but it still consists of a few top-level statements only.

hello-world-4/Program.cs

```
Console.WriteLine("What is your name?");
var name = Console.ReadLine();

if (string.IsNullOrWhiteSpace(name)) {
    name = "Friend";
} else {
    name = name.ToUpper();
}

var greeting = string.Format("Hello {0}! How are you?", name);
Console.WriteLine(greeting);
```

Readers are encouraged to read each example source code, e.g., from top to bottom, even if it may not make much sense. You will recognize the things that you already know, and the things that you do not. That is a very important part of the learning process.

4.2.1. Explanation

The program asks the user for his/her name, and it uses the name to personalize the greeting.

Let’s try running the program:

```
dotnet run
```

It prints out the following "question":

```
What is your name?
```

If you "answer" it with, say, "joe", then it prints out the following personalized greeting to the terminal.

```
Hello JOE! How are you?
```

The whole "conversation" may look like this:

```
What is your name?
Jill                          ①
Hello JILL! How are you?
```

① The input you provided on the terminal. You type, say, "Jill", and press Enter (a newline). Then, the program reads your input and does what it is instructed to do next.

4.2.2. Grammar

This sample code is mostly based on what we have already covered so far.

There are `var` declarations (lines 2 and 10). There is an `if`/`else` statement (lines 4-8). We use the `Console.WriteLine()` static method from the `System` namespace to print out texts to the terminal (lines 1 and 11). As stated, the `using` declaration for the `System` namespace is implicit.

In the second line, although we have not seen `ReadLine()` before, we can easily guess that it is a static method defined in the `Console` class (based on its syntax/form).

The empty lines are mostly ignored by the compiler. They are often used to increase the readability (lines 3 and 9).

> We could have added thousands of empty lines and it still would not have made much difference to the compiler. However, an excessive use of empty lines, or white spaces in general, could, and will, reduce the readability. ☺

This example code reads an input from the console, or "stdin", rather than getting it from the command line argument.

In line 1, it first prints out a "question" as a prompt to the user. He/she knows at this point that an input is expected, in particular, the user’s name. The program then waits for the user input.

There are a number of ways that a C# program can process user input. The example code uses another static method `ReadLine()` of the `Console` class from the `System` namespace. The `Console.ReadLine()` method returns the user input as a string. We use the returned value to initialize the local variable `name`. Therefore, the type of `name` is `string`. (More precisely, it is a "nullable string". We will defer the discussion on the nullable reference types until later in the book.)

If the user does not type anything (and just press the Enter key), then the returned value might be an empty string. We can use the static method `IsNullOrEmpty()` from the `System.String` class on .NET to determine if any non-empty string has been inputted. The sample code uses a generalized version `string.IsNullOrWhiteSpace()`. It evaluates to `true` if the given string argument (`name` in this case) is `null` or empty or if it is just white spaces only like `"      "`.

> `string.IsNullOrWhiteSpace(name)` is essentially equivalent to `(name is null || name.Trim() == "")`. The `Trim()` method removes all white spaces in the beginning and end of the string, if any. Note the difference in syntax in calling these two methods, `string.IsNullOrWhiteSpace(name)` vs `name.Trim()`. `Trim()` is an "instance method".

If no "valid" input string has been provided, then we simply set the name to `"Friend"` (the `if` block). If a non-empty name has been provided (after removing all the leading and trailing spaces), then, just for illustration, we change the name to all uppercase letters (the `else` block). For this, we use an "instance method" `ToUpper()` from the `name` object.

Note the syntax. The method `ToUpper()` is called on `name` using the "dot notation". The method `ToUpper()` is defined in the `System.String` class, but it is used on an "instance" of the `String` class, namely, `name` in this example. Methods like this are called "instance methods" as just stated.

> We will discuss the difference between the static methods/properties and the instance methods/properties, and when to use one vs the other, throughout this book. But, as indicated, we defer the full discussion of the OOP, object oriented programming, to Part II.

The converted `name` value is then assigned back to the same variable `name`. This is a common idiom since we will not need the old value of `name` once the uppercase conversion is done.

(The old value of `name`, e.g., the value originally read from the input, line 4) is no longer accessible from the program, after this `if/else` statement, and it can be " garbage-collected". More on this later. One other thing to note is that, as stated, certain statements like `if/else` that use the "blocks" do not end with semicolons.)

Next, a personalized greeting is created using the static method `string.Format()`:

```
var greeting = string.Format("Hello {0}! How are you?", name);
```

This statement is equivalent to the following, using the interpolated string:

```
var greeting = $"Hello {name}! How are you?";
```

The `Format()` method of `System.String` takes one or more arguments. The first argument, a string, is a format specifier, which includes the placeholders like `{0}` and `{1}`, etc. These placeholders are replaced by any subsequent arguments, in the order, starting from zero `{0}`.

In this example, there is only one placeholder and one additional argument. The result will be a single string. If `name` is `"JOHN"`, then `greeting` will be `"Hello JOHN! How are you?"`

> Some people prefer doing the string formatting using `string.Format()` (say, rather than using the newer string interpolation syntax). This is really a matter of taste, and preference, but note one drawback of using the string formatting.
>
> When we update the program, the formatting and the argument list may go out of sync. One can easily imagine how it can happen. And, *it does happen*. Using the `$`-string interpolation is more robust to the potential errors like this.

The value of `greeting` is then printed to the console using the `WriteLine()` method as in the previous hello world examples, and the program terminates (because there are no more statements to run).

An interesting thing about `Console.WriteLine()` is that it is very similar to the `string.Format()` method in terms of the way it accepts arguments. For example, we could have simply called `WriteLine()` as follows:

```
Console.WriteLine("Hello {0}! How are you?", name);
```

instead of calling `string.Format()` first, as in the sample code:

```
var greeting = string.Format("Hello {0}! How are you?", name);
Console.WriteLine(greeting);
```

These two code segments are more or less equivalent to each other.

4.3. Code Reading - Hello World 4 Redux

Finally, let’s look at the following example:

hello-universe-4/Program.cs

```
class Program {
    static void Main() => Console.WriteLine(
        string.Format("Hello {0}! How are you?", ReadName())
    );

    static string? ReadName() {
        Console.WriteLine("What is your name?");
        var name = Console.ReadLine();

        if (string.IsNullOrEmpty(name)) {
            name = "Friend";
        } else {
            name = name.ToUpper();
        }

        return name;
    }
}
```

This example code is semantically equivalent to "hello-world-4".

> *Is this "Hello World" thing ever gonna end?* ☺
>
> This is the last one. It’s a promise! ☺ But, remember, it is not about "hello world" or any particular examples that we use in this book. We are learning the C# language, and its idiomatic modern programming styles. Note that the book begins rather slow, but it

> quickly starts to progress pretty fast. Better fasten your seatbelt! ☺

4.3.1. Explanation

This program behaves in exactly the same way as its earlier counterpart, hello-world-4. The program asks the user for his/her name as before, and it uses the user input to personalize its greeting. Let’s run the program:

```
$ dotnet run                 ①

What is your name?
joe                          ②
Hello JOE! How are you?
```

① The command.

② The user input provided on the terminal.

4.3.2. Grammar

As stated, the top-level statements are effectively a part of an implicitly defined `Main()` static method (of an implicitly defined, or compiler-generated, class).

In this example, some of the code has been put into a different static method, `ReadName()`. The `Main()` method, which uses the expression body syntax, calls this method. In general, using modular structures like this is preferred (even though we may never intend to re-use the `ReadName()` method in other parts of the program). The code written this way is generally easier to read, and easier to maintain over time.

One thing to note is that the `ReadName()` static method returns a value of type `string?` (say, instead of `string`). This is a "nullable string" type. We will come back to this topic of the nullable reference types later in the book.

The hello-world-4 program could have been written using a separate static method as well. For example,

```
var name = ReadName();
var greeting = string.Format("Hello {0}! How are you?", name);
Console.WriteLine(greeting);

static string? ReadName() {
    Console.WriteLine("What is your name?");
    var name = Console.ReadLine();

    if (string.IsNullOrEmpty(name)) {
        name = "Friend";
    } else {
        name = name.ToUpper();
    }

    return name;
}
```

Except for some minor differences like using an intermediate local variable, etc., this is equivalent to both hello-world-4 and hello-universe-4.

Moving forward, we will use this style of code for the "main" part of the programs in this book. As stated, this is mostly a matter of preference, and it is not required. If you prefer the more traditional style of explicitly using the `Main()` method in a class, then that should be fine as well.

> In the modern C#, one can also define the program's `Main()` method in an interface.

Summary

In these beginning four lessons, we covered some basics of input and output handling in a C# program. This kind of user interaction is often known as the " command line interface", or the *CLI* for short.

In particular, in this lesson, we learned how to use the `Console.ReadLine` static method to read the user input from the command line.

In addition, we also reviewed a few static and instance methods defined on the builtin `string` type. The static methods are used with the associated type, whereas the instance methods are used with an object of the type.

Although there are still a lot that we need to learn, these "hello world" lessons introduce some of the most important and essential concepts of the C# programming language. The readers can now start programming in C# on their own, especially using the "top-level statements".

4.4. Exercises

1. Review the sample code of "hello-world-4". Close the book, and write a program that does more or less the same thing. The program asks the user for his/her name and uses it to print out a customized greeting, after changing the name to all uppercase. For example, write "!Hola JESUS!" to the console when the user input is "Jesus".
2. Write a version of Hello World program with the following requirements:
    - If the program is run with a command line argument, use it as a name as in "hello-world-3" or "hello-universe-3" of the previous lesson.
    - If a full name is provided ("first_name last_name"), then use the full name (including the space).
    - If the program starts without a command line argument, ask for the user’s

name as in "hello-world-4" or "hello-universe-4".

- Print "Hi <your name>. Now you are a C# programmer!" to the terminal.

To clarify, if you run your program this way, for example,

```
dotnet run John Smith
```

The output should be something like this:

```
Hi John Smith. Now you are a C# programmer!
```

If you run your program without any arguments, e.g., `dotnet run`, then the program will behave differently.

Author’s Note

"Thought Programming"

The author loves books. He owns thousands of Kindle books (although he has read only a tiny fraction of them 😀).

The Art of C# - Basics: Introduction to Programming in Modern C# - Beginner to Intermediate is a book for reading. Keep it on your night stand. Keep it on your coffee table. Take it to lunch.

You do not need a computer to read this book. Read, and if you need to practice, then do it in your head. Like "thought programming". As in a "thought experiment".

Obviously, this is an oxymoron. But, it is possible. And, this is a much better alternative to making excuses and postponing. You may say, "Oh, I don’t have access to computer right now. I’ll do it later". And, you may never do it. Just do the "thought programming" when you need to do programming.

In fact, "thinking" is the most important part of programming.

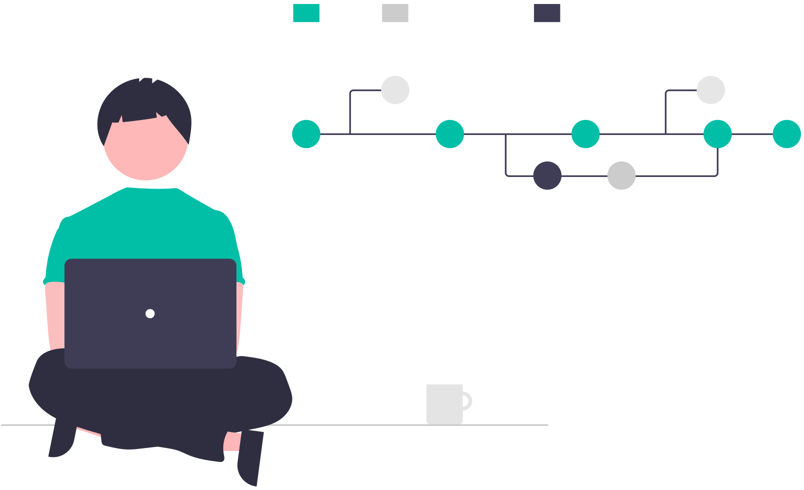
If you are just starting out, then there will be a lot of people telling you what to do, and what not to do, to become a "good programmer". They will try to make you a programming technician. They will try to make you an "assembly worker" (a la Henry Ford). They will look at you funny if you insist that programming is a creative work.

Don’t fall for that. 😉

Chapter 5. Simple Arithmetic

5.1. Agenda - Numeric Literals, Arithmetic and Bitwise Operators, Tuples, tuple Deconstruction

We will start exploring some basic concepts of the C# programming language. In particular, we will learn how to do simple arithmetic, and other operations, in C# in this lesson.



If you are familiar with the basics of programming (in any language), and if you find any of the lessons too slow moving, then you can skip them without losing too much continuity. This lesson, in particular, has been included here for completeness, for the beginners with no or little experience in programming.

5.2. Code Reading - Simple Operations in C#

This sample code illustrates various operations using primitive, or built-in, types.

simple-arithmetic/Program.cs

```
using static System.Console;

var str = "C#" + "Language";
WriteLine($@"""C#"" + ""Language"" = ""{str}""");

var sum = 1 + 2;
WriteLine($"1 + 2 = {sum}");

var sub = 5L - 2;
WriteLine($"5 - 2 = {sub}");

var prod = 1.0 * 5.0;
WriteLine($"1.0 * 5.0 = {prod}");

var div = 8.0F / 3.0;
WriteLine($"8.0 / 3.0 = {div:.##}");

var money = 10.25M;
var dollars = decimal.Floor(money);
var cents = decimal.Subtract(money, dollars) * 100;
WriteLine($"$10.25 = {dollars} dollars and {cents:0} cents");

var (numer, denom) = (7, 2);
var (quotient, remainder) = (numer / denom, numer % denom);
WriteLine($"{numer} / {denom} = {quotient}");
WriteLine($"{numer} % {denom} = {remainder}");

var boolAnd = true && false;
var boolOr = true || false;
WriteLine($"t && f = {boolAnd}; t || f = {boolOr}");

var b1 = (byte)0b10;  // 00000010
var b2 = (byte)0b110; // 00000110
var bitAnd = (byte)(b1 & b2);
var bitOr = (byte)(b1 | b2);
var bitShift = (byte)(b2 << 2);
WriteLine($"b1 & b2 = {Fb(bitAnd)}; b1 | b2 = {Fb(bitOr)}; b2 << 2 = {Fb(bitShift)}");

// Returns an 8-bit pattern as a string for a given byte argument.
static string Fb(byte b) => $"{System.Convert.ToString(b, 2)}".PadLeft(8, '0');
```

5.2.1. Explanation

The "main program" includes a series of statements which perform some basic operations. It prints out the results using `Console.WriteLine()`.

If you run the program as before using the `dotnet` CLI tool,

```
dotnet run
```

It prints out the following output to the console:

```
"C#" + "Language" = "C#Language"
1 + 2 = 3
5 - 2 = 3
1.0 * 5.0 = 5
8.0 / 3.0 = 2.67
$10.25 = 10 dollars and 25 cents
```

```
7 / 2 = 3
7 % 2 = 1
t && f = False; t || f = True
b1 & b2 = 00000010; b1 | b2 = 00000110; b2 << 2 = 00011000
```

5.2.2. Grammar

A computer is a machine that "computes". At the most fundamental level, computers deal with numbers. Binary numbers. Everything that the computer stores and processes is `0`s and `1`s.

So, what does, for example, `00110011` mean?

Suppose that we have a series of 0's and 1's in a particular location in memory, for instance. What do these numbers mean?

This is where the "types" come in. Based on the type of the value at a certain memory location, we can interpret what those numbers mean.

If a byte in memory has a value `01000001` and if its type is `byte`, then it is `A` (or, a number 65). If the byte is a part of a 32 bit integer type value, then it could mean a very different thing. If this byte is a part of a value of a string type, then it could mean `A` or something completely different (based on the byte alignment, etc.).

"Types" are of the fundamental importance in programming, especially for the " statically typed languages" like C#.

We deal with a number of different "builtin types", or "primitive types" in this sample code.

- String type: Lines 3~4
- Integer types: Lines 6~10, 23~26, 32~37
- Floating point number types: Lines 12~21

- Boolean type: Lines 28~30

> C# has a "unified type system". Every type inherits from the base `object` type (which corresponds to `System.Object` in .NET). This is true for both value types and reference types. This is true for both builtin types and user-defined types.

The rules of basic operations in C#, such as addition and multiplication, are essentially identical to those of almost all of the C-style languages such as C/C++, Java, and Go.

If you have been programming in any of these languages, then there is really not much new to learn in C#, as far as these basic arithmetic operations go.

Line 3 of the sample code implicitly declares a variable `str` using the keyword `var`. Since the right hand side is a string concatenation expression which evaluates to a string, the type of `str` is `string`. In fact, it will be inferred as the "nullable string" type (denoted as `string?`) as will be further discussed later in the book.

In this case there is little difference. But, if we needed the variable to be the non-nullable string type, then we could have explicitly declared it as `string`.

```
string str = "C#" + "Language";
```

To print the value of `str`, we use the `$`-string interpolation.

The `@` symbol prefix indicates that it is a "raw string literal". In a raw string, many characters that should have been "escaped" in a normal string literal (including the `$"…"` strings) can be included without escaping.

The `$@"…"` syntax indicates that it is a raw string and it uses the string interpolation. The double quote char (`"`) in a raw string still needs to be escaped using the two double-quote `""` syntax.

Note that we use the `using static` declaration and use the short form `WriteLine()` instead of the longer qualified name `Console.WriteLine()` in this program.

> ℹ One thing to note about `string` is that the `string` type is rather special in C# (and in many other similar languages) in that it is a reference type, and yet it mostly behaves like a value type. We will further discuss this later in the book.

In line 6, since the right hand side is an expression with integer literals, the new variable `sum` is of type `int`.

C# supports a number of different integer types, each of which corresponds to a specific System class in .NET. Integers are all value types.

sbyte

Signed 8-bit integer. `System.SByte`.

byte

Unsigned 8-bit integer. `System.Byte`.

short

Signed 16-bit integer. `System.Int16`.

ushort

Unsigned 16-bit integer. `System.UInt16`.

int

Signed 32-bit integer. `System.Int32`.

uint

Unsigned 32-bit integer. `System.UInt32`.

long

Signed 64-bit integer. `System.Int64`.

ulong

Unsigned 64-bit integer. `System.UInt64`.

nint

Signed 32-bit or 64-bit integer depending on platform.

nuint

Unsigned 32-bit or 64-bit integer depending on platform.

Note that there are 8, 16, 32, and 64 bit integer types and that there are signed and unsigned versions for each of these types. In addition, C# provides two "native" integer types, signed `nint` and unsigned `nuint`. On a 32-bit machine, their size is 32 bits. On a 64-bit machine, their size is 64 bits.

The integer literals are by default of the `int` type. If one needs a different type, various suffixes can be used. Suffixes `l` or `L` indicates that the literal is of type `long`. Suffixes `u` or `U` indicates that the literal is of type `unsigned`. These suffixes can be combined.

Alternatively, one can use an explicit casting, using the `(type_name)` syntax. For instance,

```
var a = 1UL;
var b = (ulong)1;
ulong c = 1;
```

All three variables, `a`, `b`, and `c`, have the same type, `ulong` (unsigned long), in this example.

In line 9, the right hand side expression evaluates to `long` (because one of the operands, `5L`, is of type `long`, which is "wider" than `int`), and hence the type of `sub` is `long`.

Line 12 of the example code introduces the floating point number operations. On the right hand side, two numbers are multiplied, and its result is used to initialize a variable `prod`. An implicit variable declaration is used using `var`.

C# includes three different floating point types, each of which again corresponds to a specific System class in .NET.

float

4 bytes. `System.Single`.

double

8 bytes. `System.Double`.

decimal

16 bytes. `System.Decimal`.

The floating point numeric literals are by default inferred to be of type `double`. One can use the suffixes `f`/`F` and `m`/`M` to explicitly make them types `float` and `decimal`, respectively. One can also optionally use `d`/`D` for the values of `double`.

The type of `prod` (line 12) is `double`. The type of `div` (line 15) is also `double`. Although the first operand `8.0F` is `float`, the second operand `3.0` is `double` and hence the result of the operation is `double`.

Note in this example that the expression, `8.0` divided by `3.0`, has no exact representation on the computer. The result is `2.66666666666666`…. The notation `{div:.##}` means that the output should be no more than two digits below the decimal point. The output in this case is `2.67`, as shown above.

One thing to note is that the floating point number calculations on computers have finite precisions. This is generally true, not just in cases like this example. The real numbers are only approximately correct, and one has to be mindful when dealing with the floating numbers.

C#'s `decimal` type provides higher precision (e.g., by using 16 bytes, as compared to the 4 byte single precision). The decimal type numbers can be used for financial calculations, for example.

The type of `money` (line 18) is `decimal`. The `decimal` type includes some static convenience methods to facilitate high precision calculations. For example, the example code uses two such methods, `decimal.Floor()` and `decimal.Subtract()`. The notation `{cents:0}` instructs to print out only the number above the decimal point. Alternatively, we could have converted `cents` (as well as `dollars`) to `int`.

Lines from 23 to 26 show other integer computations, namely integer division and modulo operations.

In math, an integer division can produce a real number result, as in $3/2 = 1.5$. In C#, an integer division produces an integer number with the same type as their operands (or, the "wider" of the two if the two types are different).

Dividing integer `7` with integer `2` is integer `3`, as can be seen from the output of line 25. The modulo operation produces the remainder of an integer division as a result, which is integer `1` in this particular example (the output of line 26).

> In the "loosely typed" languages, this may not be true. For example, in Javascript, there are no separate types like `int` or `float`. There is only one number type, `number` (which corresponds to `double` in C#). In Python, there are two different division operators, one (`//`) for the integer division and the other

> (`/`) for the floating point division, the latter of which produces a real number regardless of their operand types.

We have not discussed any complex or composite types yet, but .NET has a type called "ValueTuple" (as well as the "Tuple" type).

The right hand side expression of line 23 is a syntax representing a value of type `ValueTuple` in C#. `(7, 2)` is a tuple (or, a tuple literal) of two `int` values, `7` and `2`.

Each of these values are then used to initialize two variables, `numer` and `denom`. This operation is called the "deconstruction" (or, "destructuring" or "unpacking"). Instead of assigning the tuple value to a single variable and accessing each item later, we just assign them in one go.

This statement is equivalent to the following:

```
var t = (7, 2);
var numer = t.Item1;
var denom = t.Item2;
```

Or, more simply

```
var numer = 7;
var denom = 2;
```

We do it in one statement in the sample code, instead of using two or three, using an anonymous tuple `(numer, denom)`. ("Anonymous" in that the tuple variable itself does not have a name.)

The statement in the next line (line 24) has the same structure. A two-item tuple is created on the right hand side, and each of the items are assigned to new variables `quotient` and `remainder` through deconstruction.

This statement is equivalent to

```
var quotient = numer / denom;
var remainder = numer % denom;
```

Note that, in assignment or initialization, the expressions on the right hand side of a statement are always computed first before the assignment or initialization.

We can, for example, use tuples to easily swap two numbers in C#.

```
var (x, y) = (1, 2);
(x, y) = (y, x);
Console.WriteLine($"x = {x}, y = {y}.");
```

What would be the output of this code? How would you implement this without using a tuple?

> The item access via the predefined properties, `Item1`, `Item2`, `Item3`, … is pretty opaque. It is not entirely clear what `Item1` is, etc., in this particular case. On the other hand, the deconstruction syntax provides a better readability.

The statements in lines from 28 to 30 demonstrate the Boolean, or logical, operations. Namely, Boolean AND `&&` (line 28) and Boolean OR `||` (line 29). The Boolean operations in C# can be done only with Boolean type variables and expressions. There is no implicit type conversion between the `bool` type and other numeric types in C#.

The builtin `bool` type in C# corresponds to the `System.Boolean` class in .NET.

As the output of `WriteLine()` in line 30 shows, `true && false` is `false` whereas `true || false` is `true`. For completeness, `true && true` is `true` and `true || true` is `true`. `false && false` is `false` and `false || false` is `false`.

Although we do not include in the example code, C# also has a logical "NOT" operator `!`. It "negates" the Boolean value. That is, `!true` is `false` and `!false` is `true`.

> An "operator" is really a "function" with a special syntax in a programming language. C# has *a lot of* different operators. In fact, there are so many that this book would *explicitly mention* only a few. Some operators have different meanings, or semantics, depending on the context of their use. In C#, we can even define our own "custom operators" (known as the "operator overloading").
>
> A class of operators that we do not explicitly define in this book, although we extensively use, are so-called "comparison operators". They are binary operators (those that take two operands, or arguments), and they produce boolean values. They are common across many different programming languages.
>
> For example, the equality operator `==` returns `true` if the two operands are "the same" in some predefined definition. For example, for two integer values the value equality is used. That is, `3 == 3` evaluates to `true`.
>
> One other thing to note is that the use of these operators may seem rather confusing to the people new to programming. `=` (the math equality symbol) is used for "assignment", among other things. On the other hand, `==`, which looks so similar, is the equality operator. The fat arrow operator `=>` looks rather similar to the inequality operators, `<`, `<=`, `>`, and `>=`. C# also uses operators like `!`, `?`, and `??`, for instance, in many different contexts with

> different meanings.
>
> Unfortunately, you will have to get used to these notations. It takes time. Just remember, it's all about "familiarity".

The final segment of the sample program illustrates the "bitwise operations".

First, numbers, or numeric literals, can be represented with decimal numbers or numbers with a different base. In particular, C# supports binary, octal, and hexadecimal number literals. Hexadecimal numbers start with `0x`. For example, `0x1` is 1. and `0x10` is 16 in the decimal representation. Octal numbers start with `0` followed by another number, and Binary numbers start with `0b`.

The statements in lines 32 and 33 declare two `byte` variables, `b1` and `b2`, and initialize them with two numbers, 2 (`0b10`) and 6 (`0b110`), respectively. Note that we explicitly cast the (int) numbers before assigning them to `byte` variables.

The comments in those lines show the bit pattern of each number (since we are going to demonstrate the bitwise operations).

Since the leading zeros are ignored after the integer base prefix (that is, `0x10` is the same as `0x00010` as far as C# programs are concerned), we could have written them as follows:

```
var b1 = (byte)0b00000010;
var b2 = (byte)0b00000110;
```

Or, even

```
var b1 = (byte)0b_0000_0010;
var b2 = (byte)0b_0000_0110;
```

Underscores in integer literals are ignored. It’s not entirely obvious from this simple example, but including underscores in numbers can increase the readability and reduce the chance of inadvertent errors.

In this particular example, note that a set of 4 digits in a binary number corresponds to 1 digit in a hexadecimal number.

> C# supports the C++/Java style comments. `/*` … `*/` is a multi-line comment. Anything between `/*` and `*/` is ignored by the compiler. `//` … is a single line comment. Anything after `//` in the same line is ignored.
>
> Although we call `/*` … `*/` the multi-line comment, by convention, it is more versatile. You can comment out one line, or you can even comment out a part of a line, using this syntax. In general, however, most programmers prefer the "single line comment" syntax in most cases. And, they use it for multi-line commenting as well. Just use `//` in front of every line that needs to be a comment.
>
> The "names" can be deceiving. ☺

Lines 34 and 35 demonstrate the bitwise AND and bitwise OR operations, respectively. In the bitwise AND or OR operations, the corresponding bits of the two numbers, or bytes, are operated on independently of the other bits (hence the name "bitwise operations").

`1 & 1` results in `1`. Likewise, `1 & 0` is `0`, `0 & 1` is `0`, and `0 & 0` is `0`. Also, `1 | 1` is `1`, `1 | 0` is `1`, `0 | 1` is `1`, and `0 | 0` is `0`.

The statement of line 36 demonstrates the (left) bit-shift operation. It shifts each bit of the given byte (the first operand) to the left by the second operand. That is, `0b_1110_0010 << 1` becomes `0b_1100_0100`. Note that it does not "wrap around".

Rather, it fills the right-most positions with zeros.

The same holds true with the right bit-shift operator `>>`. It just moves all bits to the right (with no wrapping).

The code sample includes a small helper function, `Fb()` (for "format byte"), to print out the bit patterns of the byte variables. Note that this top-level static method is defined with the "expression body".

As for what exactly this method does, and how it does it, we will leave it to the readers as an exercise. It is "optional". (You may have to refer to the (online) API documentation, which may require a skill or knowledge that this book does not explicitly teach.)

Summary

We learned how to do basic operations in C#. There are integer and float operations. Boolean operations. And, bitwise operations.

C#, and .NET, supports a number of different integer and floating types. Integer literals can be represented in base 2, base 8, base 16 as well as in base 10.

String literals and variables can be used just like they are numbers or values. They can be added (e.g., string concatenation) and they can be compared with each other using the equality operator (`==`), among other things.

We also introduced the "raw string" syntax in this lesson.

5.3. Exercises

1. Write a program that takes two integer command line arguments and prints out four numbers to the console, the addition, the subtraction, the multiplication, the division of the two integer numbers. If the second number is `0`, then it prints

out `NaN` instead of the division number (which does not exist). Note that the command line arguments are read as `strings` (even when they look like numbers) and you will need to convert the string variables to integers. Use the `Int32.Parse()` static method for that. We will cover this in more detail later in the book.

Author’s Note

Software Stack

Building software is not unlike building a skyscraper, or a pyramid.

There are things that go near the ground, and there are things that go near the top. Viewing software as a vertical stack of blocks is a very useful metaphor. Sometimes we look at things from top to bottom, and sometime we look at things from bottom to top.

In C#, the building blocks are "custom types" and "static methods".

When you use a class property or method or a static method from the (dn) standard library, for example, you are putting your block on top of the standard library blocks.

The closer to the ground, the building blocks do smaller but more generic tasks. The blocks in the higher up do broader but more specific tasks.

At the top of the pyramid is the `Main()` method of the main class, or interface, of your program. The "main" class in a C# program is no different from any other types in C#.

In C#, all types and static methods in a program are all compiled together as if they are all from one giant source code file. There are no strict linear, or directional, dependencies within a program. Any entity can depend on, or use, any other entity in the same program.

In that sense, the true building block in a C# program is an "assembly", and not its components like types and static methods. We are building an executable assembly, for instance, on top of other library assemblies.

But, regardless, it is a good practice to view each component in an assembly as a small block. And, to think about the "dependencies" between these small blocks.

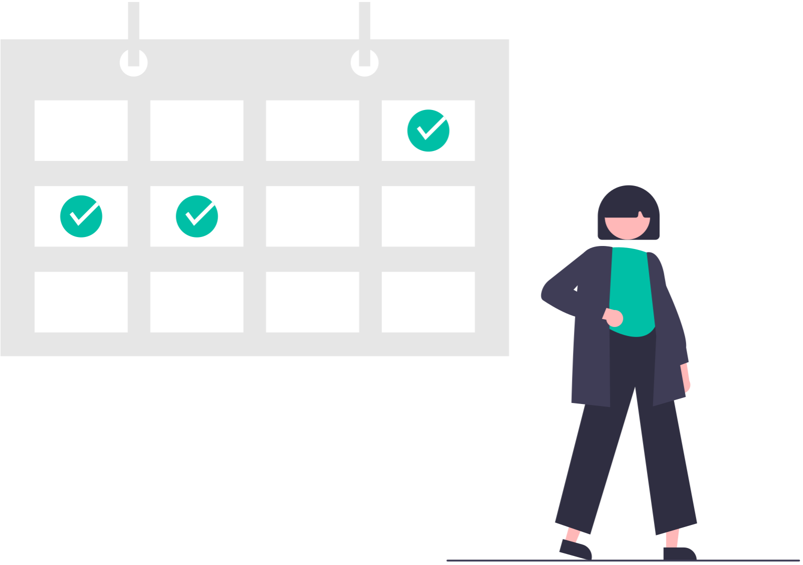
A C# program that is one giant soup of components, each of which depend on the rest, will be hard to understand, hard to troubleshoot, and hard to maintain.

It is best to treat building a software like building a "pyramid".

Chapter 6. Leap Year

6.1. Agenda - Delegates, Arrays, `foreach` Loops, `if` Conditional Statements

We will explore some more basic concepts of the C# programming language in this lesson.



6.2. Code Reading - Check If the Given Year is a Leap Year

Here’s a C# project file.

leap-year/leap-year.csproj

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

As indicated earlier, we will *always* enable the nullable context in all sample code in this book (whether it is relevant or not to the specific example).

Let's create a program that checks if a given year is a leap year. Here's the "main" program:

leap-year/Program.cs

```csharp
using Example;

var isLeapYear = LeapYear1.IsLeapYear;

var years = new[] { 1900, 1984, 2000, 2022 };
foreach (var year in years) {
    var ans = isLeapYear(year);
    Console.WriteLine($"Is {year} a leap year? {(ans ? "Yes" : "
No")}");
}
```

We will look at three different implementations, which all use the same method signature, as defined here:

leap-year/LeapYear.cs

```
namespace Example;

delegate bool IsLeapYear(int year);
```

We will discuss "delegates" shortly.

Here’s the first implementation:

leap-year/LeapYear1.cs

```
namespace Example;

static class LeapYear1 {
    internal static bool IsLeapYear(int year) {
        if (year % 4 == 0) {
            if (year % 100 == 0) {
                if (year % 400 == 0) {
                    return true;
                } else {
                    return false;
                }
            } else {
                return true;
            }
        } else {
            return false;
        }
    }
}
```

The second implementation:

leap-year/LeapYear2.cs

```
namespace Example;

static class LeapYear2 {
    internal static bool IsLeapYear(int year) {
        if (year % 400 == 0) {
            return true;
        } else if (year % 100 == 0) {
            return false;
        } else if (year % 4 == 0) {
            return true;
        } else {
            return false;
        }
    }
}
```

And, the third implementation:

leap-year/LeapYear3.cs

```
namespace Example;

static class LeapYear3 {
    internal static bool IsLeapYear(int year) {
        if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0) {
            return true;
        } else {
            return false;
        }
    }
}
```

Note that all three implementations use static classes. A static class cannot be instantiated, and it is only used as a container for static methods. C# does not support top-level functions, and static methods are the closest thing to functions.

One another thing to note from this sample code is that `namespaces` are "open" in that you can use the same namespace block in multiple files (within a program). This is not true for `classes`, for instance. (An exception is the "partial class", and the "partial method", but we will not discuss the use of partial classes in this book.)

> It’s not really recommended, or even necessary, to use many small files, or to put different classes in different files, etc. The code samples in this book are mainly organized for easy presentation in a book format.

6.2.1. Explanation

The program contains three different implementations of `IsLeapYear()`.

In the included example of the "main" program, the first version `LeapYear1.IsLeapYear()` is hard-coded.

If you run the program, for example, using the `dotnet` CLI tool,

```
dotnet run
```

It prints out the following output to the console:

```
Is 1900 a leap year? No
Is 1984 a leap year? Yes
Is 2000 a leap year? Yes
Is 2022 a leap year? No
```

If you want to use the second version instead, for instance, then you can assign the method name, `LeapYear2.IsLeapYear`, to the variable `isLeapYear` in the beginning of the program.

```
IsLeapYear isLeapYear = LeapYear2.IsLeapYear;
```

6.2.2. Grammar - Delegates, Arrays, Conditional Statements

Methods in C# can be given an explicit type. It is called a `delegate`.

A delegate is a type that represents the references to certain methods with a particular parameter list and return type.

For instance, in the sample code, we declare a delegate named `IsLeapYear` (in the "file-scoped namespace" `Example`).

```
delegate bool IsLeapYear(int year);
```

This statement declares a "delegate type", using the keyword `delegate`, which represents a method which takes an argument of type `int` and returns a `bool` value, and gives it a name `IsLeapYear`. (`IsLeapYear` is the name of this type, just like `int` or `string`.)

As stated, as of C# 10.0, the use of the file-scoped names syntax is preferred since it reduces the need for indentation. This source code could have been written as follows, using the more traditional block namespace:

```
namespace Example {
    delegate bool IsLeapYear(int year);
}
```

Note the indentation within the namespace block. This is sometimes called the "horizontal waste". It is just one line in this example, but one can easily imagine how much unnecessary space we can remove by using the file scoped namespaces in large programs (e.g., with nested namespaces, etc.).

When you instantiate a delegate, you can associate its instance with any method with a compatible signature (e.g., the same argument types and the return type). A delegate instance may encapsulate either static or instance method. You can invoke (or call) the method through the delegate instance.

The delegates are mainly used to pass methods as arguments to other methods. The delegate types are reference types.

In our example, we merely use it as a common type for all three different implementations of `IsLeapYear()`, which all have the same function signature.

This statement in the main program "instantiates" a delegate `IsLeapYear`, with a variable name `isLeapYear`, and initializes it with a value `LeapYear1.IsLeapYear`, which has a signature compatible with the delegate `IsLeapYear`.

```
var isLeapYear = LeapYear1.IsLeapYear;
```

Note that we could not have used an implicit `var` declaration here if we had used C# version 9.0 or earlier. The compiler would not have been able to guess what (delegate) type to use if we had used the implicit `var` declaration. This has been improved since C# 10.0.

If we want to use a method as an object (e.g., so that we can pass them as arguments to other methods, etc.), then we need to declare an appropriate delegate type first. And use an instance of that type as a "delegate" of the method. (Hence the name.)

.NET includes a number of predefined delegate types, making it much easier and more natural to use delegates in place of methods. The "main program" could have been written as follows, for instance:

```
using Example;

CheckLeapYears(LeapYear1.IsLeapYear);

static void CheckLeapYears(IsLeapYear isLeapYear) {
    int[] years = new[] { 1900, 1984, 2000, 2022 };
    foreach (var year in years) {
        var ans = isLeapYear(year);
        Console.WriteLine($"Is {year} a leap year? {(ans ? "Yes" :
"No")}");
    }
}
```

In this case, we do not explicitly instantiate a delegate of type `IsLeapYear`, and just use the method `LeapYear1.IsLeapYear` as if it is a delegate, as an argument to the `CheckLeapYears()` method.

This is possible because the method `CheckLeapYears()` accepts a variable of the delegate type `IsLeapYear`, and the implicit conversion happens in the method call (because the method signature of `LeapYear1.IsLeapYear` is compatible with the delegate type). Methods in C# do not have (intrinsic) types. We need to use delegates, explicitly or implicitly.

> Throughout this book, we will informally use the term the "main program" to refer to the part of a program, in a single source file (typically, named `Program.cs`), that uses the top-level statements.

> The top-level statements are part of an implicitly defined `Main()` static method, which is the entry point to a C# program.
>
> As stated, the top-level statement syntax is a relatively new feature in C#, and in many ways we are blazing the trail. ☺

The sample code of this lesson also introduces a few important, and basic, concepts in the C# programming language.

First, it uses an array data type. We briefly mentioned it in the previous lessons, in the context of the command line argument `args`, and if you have some experience with other (C-style) programming languages, then you should be more or less familiar with the arrays.

An array type essentially lets you store multiple variables of the same type in a single data structure. You declare an array by specifying the type of its elements with a pair of empty square brackets. For instance,

```
int[] years;
```

In this case, `years` is a variable of an array type which can store multiple values of type `int`. An array type inherits from `System.Object` just like any other types in C#, builtin or user-defined.

An array variable can be initialized using the "array initializer" syntax. For example, in sample code, we initialize the variable `years` as follows:

```
var years = new[] { 1900, 1984, 2000, 2022 };
```

This is equivalent to the following:

```
var years = new int[] { 1900, 1984, 2000, 2022 };
```

Note the syntax of the right hand side expression. It starts with a `new` operator followed by the array type, `int[]` in this case. The initial values are included in a pair of curly braces. This array variable `years` is being initialized with 4 elements of type `int`.

The expression `new int[] { /* … */ }` can be abbreviated as `new[] { /* … */ }` because the element type can be inferred from the initial values.

Or, they are more or less equivalent to the following as well (ignoring the nullability):

```
int[] years = new int[] { 1900, 1984, 2000, 2022 };
```

This can be also written as follows, omitting the `new` operator and the type entirely from the right hand side initializer expression:

```
int[] years = { 1900, 1984, 2000, 2022 };
```

Which style would you generally prefer?

An array element can be accessed (for both read and write) using the (zero-based) index notation. For example, `years[1]` will have the value `1984` (the second element). The valid indexes are the integers from `0` to `years.Length - 1`.

One nice thing about the collection data types like array is that we can easily "iterate" over their elements.

The sample code uses the "foreach loop":

```
foreach (var year in years) {
    // Do something here using the variable "year".
}
```

The `foreach` loop iterates over all the elements in a given collection, `years` in this example. The foreach block will be executed 4 times in this case (since there are 4 elements in `years`), with `year` set to `1900`, `1984`, `2000`, and `2022`, in turn.

In fact, this `foreach` statement is more or less equivalent to the following:

```
var year = 1900;
var ans = isLeapYear(year);
Console.WriteLine($"Is {year} a leap year? {(ans ? "Yes" : "No")}");

year = 1984;
ans = isLeapYear(year);
Console.WriteLine($"Is {year} a leap year? {(ans ? "Yes" : "No")}");

year = 2000;
ans = isLeapYear(year);
Console.WriteLine($"Is {year} a leap year? {(ans ? "Yes" : "No")}");

year = 2022;
ans = isLeapYear(year);
Console.WriteLine($"Is {year} a leap year? {(ans ? "Yes" : "No")}");
```

Note that the looping reduces code duplications.

The expression `ans ? "Yes" : "No"` evaluates to `"Yes"` (the value before “:”) if `ans == true`, and `"No"` (the value after “:”) if `ans == false`. This strange-looking operator “?:” is known as "*the* ternary operator" because it is (the only) operator that takes three arguments (in the C-style programming languages).

Note the parentheses, “()”, inside the curly braces, “{}”, in the `$`-string interpolation. This is needed because the argument is not a single value or a simple expression, but a multi-operand expression (which evaluates to a value).

As stated, the code inside the loop uses the delegate variable `isLeapYear` as if it is a method.

```
var ans = isLeapYear(year);
```

This is equivalent to the following:

```
var ans = LeapYear1.IsLeapYear(year);
```

Now let’s take a look at the "leap year problem" of this lesson: Given a year, determine if the year is a leap year.

A leap year has 366 days instead of 365. Interestingly, which year is a leap year is *defined algorithmically.* For instance, refer to en.wikipedia.org/wiki/Leap_year for the definition of the *leap year.*

```
If the year is divisible by 4, continue.
If the year is not divisible by 4, it is not a leap year. End.
    If the year is divisible by 100, continue.
    If the year is not divisible by 100, it is a leap year. End.
        If the year is divisible by 400, it is a leap year. End.
        If the year is not divisible by 400, it is not a leap year.
End.
```

This is an "algorithm". We translate this algorithm into a program in C#. That is essentially the `LeapYear1.IsLeapYear()` function.

If we write the above algorithm slightly differently, then it is easier to compare:

```
If the year is divisible by 4, continue.
    If the year is divisible by 100, continue.
        If the year is divisible by 400, it is a leap year. End.
        Else (if the year is not divisible by 400), it is not a leap year. End.
    Else (if the year is not divisible by 100), it is a leap year. End.
Else (if the year is not divisible by 4), it is not a leap year. End.
```

That is precisely the `LeapYear1.IsLeapYear()` static method.

This function can be rewritten in the following way, by changing the order of the `if` statements:

```
bool IsLeapYearAlt(int year) {
    if (year % 400 == 0) {
        return true;
    } else {
        if (year % 100 == 0) {
            return false;
        } else {
            if (year % 4 == 0) {
                return true;
            } else {
                return false;
            }
        }
    }
}
```

The nested `if-else` statements can be written without nesting.

```
bool IsLeapYearAlt(int year) {
    if (year % 400 == 0) {
        return true;
    } else if (year % 100 == 0) {
        return false;
    } else if (year % 4 == 0) {
        return true;
    } else {
        return false;
    }
}
```

These two implementations are exactly the same. But, this version is "flatter" and it is easier to read, and this form is generally preferred over the nested version. This is the version presented earlier, the `LeapYear2.IsLeapYear()` static method.

Now, all three Boolean expressions can be combined into a single Boolean expression.

```
bool IsLeapYear(int year) {
    if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0) {
        return true;
    } else {
        return false;
    }
}
```

You can easily convince yourself that these two implementations behave exactly the same way for all eight different conditions. (`true/false` for 3 boolean expressions yields 8. `2 * 2 * 2 = 8`.) Hence the two methods are equivalent for all possible value of the input, `year`.

The Boolean `&&` operator has a higher "precedence" than the `||` operator in C#. Hence the the parentheses around the `&&` expression can be omitted.

That's the method `LeapYear3.IsLeapYear()` presented earlier. One can further "simplify" this method using the expression body syntax

```
bool IsLeapYear(int year) => year % 4 == 0 && year % 100 != 0 || year
% 400 == 0;
```

The whole method definition is now reduced to essentially just *one line!*

Which style would you generally prefer?

The operator precedence rule determines which parts of an expression get evaluated first. For example, `2 + 3 * 4` is evaluated to be `2 + (3 * 4)`, which is `14`. This is because the multiplication operator (`*`) has a higher "precedence" than the addition operator (`+`). The explicit use of the parentheses can change the evaluation order. For instance, `(2 + 3) * 4` is evaluated to `20`.

The unary operators (e.g., `-` in front of a number) have the highest precedence. You can refer to the C# language reference for the complete list of the binary operator precedence rules, if necessary. But, you do not have to memorize these precedence rules. When in doubt, use the parentheses to make your intentions clear (e.g., to the human readers), rather than relying on some (possibly obscure) precedence rules.

Note that multiple binary operators may have the same precedence. In that case, operators of the same precedence

> associate from left to right. For example, `6.0 / 2.0 * 3.0` is the same as `(6.0 / 2.0) * 3.0`, which evaluates to `9.0`.

Before we end this lesson, there are a few more things that we will need to review.

First of all, a C# program is a collection of custom type definitions (other than the top-level statement exception). For example, the sample program of this lesson included the `LeapYear1`, `LeapYear2`, and `LeapYear3` classes as well as the `IsLeapYear` delegate.

As we will see throughout this book, a type can be defined with C# constructs like `class`, `struct`, `record`, `enum`, `interface`, and `delegate`.

Technically speaking, a static class (which cannot be instantiated) does not define a type. But, syntactically, it also belongs to these top-level elements that comprise a C# program.

As stated, the `namespaces` are only used to reduce the chance of name collision across different C# programs, and they do not play a role in the structure of a C# program.

The basic unit of the (physical) organization in .NET is an "assembly". If you compile your C# program for .NET 5, for instance, then it will be made to an assembly, an executable assembly in particular. Libraries are also built into assemblies (with `.dll` extension).

An executable or library program can use the code from other library assemblies. Code reuse and sharing happens across the assemblies. .NET provides a way to control who can access what, across different assemblies, as well as within the same assembly.

C# has a number of "access modifiers". For the top-level types that are not included in the definitions of other types, there are two kinds of access modifiers, `public` and `internal`.

The `public` types are accessible from other assemblies. The `internal` types are only accessible within the same assembly.

By default, if we do not specify an access modifier, the top-level types are considered `internal`. In our example, all three classes and the delegate are all internal, meaning that they are not meant to be shared with other programs/assemblies.

If we wanted to share the implementation of `LeapYear1`, for instance, so that other programmers could use our code, then we could have declared it as `public`:

```
public static class LeapYear1 {
    // ...
}
```

> Again, namespaces do not play any role in this regards. The namespace names are simply part of the names that we are exporting. That is, the full name of `LeapYear1` is `Example.LeapYear1`.

For the nested types (e.g., the types that are defined in another type), and for the members of a type (e.g., properties and methods), e.g., the `LeapYear1()` static method, the access control in C# is much more complicated. In fact, C# has several different access modifiers with different behaviors.

In this lesson, we used the `internal` access modifier for all three static methods, `LeapYear1.IsLeapYear()`, `LeapYear2.IsLeapYear()`, and `LeapYear3.IsLeapYear()`. Internal methods can be accessed by anybody within the same program (or, assembly).

If we want to share `IsLeapYear()`, then we would need to declare it as `public` as well, not just the `IsLeapYear` class.

The default access level for the member methods, as in this case, is `private`. The private members can be accessed only within the definition/implementation of that type. If we omitted `internal` in the declarations of the three static methods, `IsLeapYear()`, then they would not have been accessible from our "main program".

Summary

A delegate in C# is a type, and it is used to represent a group of methods, (static methods or instance methods), that have the same certain function signature.

The delegate instances can be used just like any other objects, and they can be "called" as if they are methods.

Various access modifiers like `public` and `internal` can be used to control the access levels for your types and their members.

In addition, we looked at the use of the `foreach` loop (over an array) in this lesson.

> We primarily use `delegates` in the context of the functional style programming. We do not discuss another (important) use case of delegates, namely, "events", in this book. Events are mostly used in the "event driven programming styles" (obviously ☺). If you do GUI programming, for instance, then you will end up using events more. We will leave it to the readers to find out what the event driven programming is, and what `events` are in C#.

6.3. Exercises

1. Create a new program to accept a value of `year` as a user input. Write your own `IsLeapYear()` method without referring to the sample code. Test your program with a few `year` values, and verify that your program works as

expected.

2. Create a new program to take a value of `year` as a command line argument. Do the same tests as the previous exercise.

Author’s Note

Don’t Be a Parrot

It is not uncommon to see a beginning programmer copy code from a book to a computer. Or, copy the code found on the Web to his/her computer.

Often they type the code, not even just copy and paste, and they claim that they *learn better* by actually typing.

There is no evidence for that. If anything, that will be a very inefficient way to learn programming. While an imitation is an important part of learning a new language, or a new skill, in the spirit of "listen and repeat", mindless imitations would not enhance your speaking or writing skills very much.

The author made a conscious decision not to release the sample code of this book. For one thing, it has little value. But, there are other reasons as well. In his opinion, the downloadable code samples do more harm than good.

Often the learning students download a code sample and run it on their computers. And, they think that that’s the end of it. *It works. Now, let’s move on.*

In doing so, however, they have learned very little. On the contrary, they only ended up with the false sense that they were able to "write" the same code because they compiled the code and ran it. Or, because they even "typed" the code.

Obviously, they did not *write* the code.

Although the author has asserted that this book is "for reading", if you are inclined to try out some sample code in this book, then here’s a suggestion.

1. Learn the main points of the lesson.
2. Try to understand the sample code, and what it does.
3. Then *close the book*.
4. Recall the problem which the sample code is trying to solve.
5. Create your solution to the problem.

You may, or more likely may not, end up with the same code. But, that’s perfectly all right.

If you get stuck, then refer back to the example code. Try to understand what it does, and how it does it. And then, *close the book* and try again.

Chapter 7. Multiplication Table

7.1. Agenda - Static Classes, Static Methods, Arrays, Two-Dimensional Arrays, for Loops

We will cover in this lesson some basics of array and the "classic for loops" in C#.

7.2. Code Reading - Print the Multiplication Table

Here’s the "main program", which prints out the multiplication table within a given integer range.

multiplication-table/Program.cs

```
using Example;

const int low = 2;
const int high = 10;
```

```

Console.WriteLine("Multiplication Table:");
MultiplicationTable.WriteTable(low, high);
```

The actual implementation is included in a pair of static methods.

multiplication-table/MultiplicationTable.cs

```
namespace Example;
using static System.Console;

static class MultiplicationTable {
    internal static void WriteTable(int low, int high) {
        var table = CreateTable(low, high);

        Write("        ");
        for (var v = low; v < high; v++) {
            Write($"{v,4}");
        }
        WriteLine("");

        Write("       -");
        for (var v = low; v < high; v++) {
            Write("  --");
        }
        WriteLine("");

        for (var l = low; l < high; l++) {
            Write($"{l,4}");
            Write("   |");
            for (var r = low; r < high; r++) {
                Write($"{table[l - low, r - low],4}");
            }
            WriteLine("");
        }
    }

    private static int[,] CreateTable(int low, int high) {
        var range = high - low;
        var table = new int[range, range];

        for (var l = low; l < high; l++) {
            for (var r = low; r < high; r++) {
                table[l - low, r - low] = l * r;
            }
        }

        return table;
    }
}
```

7.2.1. Explanation

We all learned the multiplication table by heart. The code generates a multiplication table within a given integer range, and it prints out the result in a certain way.

If you build and run the program from the command line, then it produces the following output:

```
        2   3   4   5   6   7   8   9    ①
    -  --  --  --  --  --  --  --  --    ②
2   |   4   6   8  10  12  14  16  18    ③
3   |   6   9  12  15  18  21  24  27
4   |   8  12  16  20  24  28  32  36
5   |  10  15  20  25  30  35  40  45
6   |  12  18  24  30  36  42  48  54
```

```
    7   |  14  21  28  35  42  49  56  63
    8   |  16  24  32  40  48  56  64  72
    9   |  18  27  36  45  54  63  72  81
```

① "Header".

② Divider between the header and the body.

③ Actual multiplication values start from this line.

> To repeat, the word "you" is not necessarily referring to *you*, the reader, in this context. The author realize that, for some people, the doctrine of "learning by doing" is so deeply ingrained into their brains that it is hard for them to open up to a new idea like "learning programming by reading".
>
> The author's advice is, *Give it a try!* ☺

7.2.2. Grammar

We have discussed the `const` and `var` variables before.

In programming in general, a variable is something that holds a value. A `const` can be viewed as a variable as well. In C#, the value of a constant is fixed at build time, and it cannot change.

In this small example, we could have just used `var` variables for `low` and `high`. But, using `const` constants is generally preferred over variable variables. In C#, however, you can only use `const` for values that are known at compile time. `const` is limited to the boolean and number types and `string`s.

Note that these values are "hard-coded" in this example. If these numbers have to be set via a program's action, e.g., by reading a user input at runtime, then they could not have been declared as `const`.

Next, the main program calls the `WriteTable()` static method on `MultiplicationTable` with parameters `low` and `high`. When the method does what it is supposed to do and returns, the program terminates.

The static class `MultiplicationTable` has two static methods, `WriteTable()` and `CreateTable()`.

The `WriteTable()` method is declared as `internal` since it is called by the main driver program. On the other hand, the `CreateTable()` method is only used within the `MultiplicationTable` class (e.g., by `WriteTable()`, line 6), and it is declared as `private`.

The `CreateTable()` method builds the multiplication table "data". It internally stores the data in an array variable, `table`, while working on the table, and it returns the complete data at the end of the execution.

C# supports a "multi-dimensional array" type, which is different from an array of arrays. E.g., `int[][]` is a (one-dimensional) array of values whose type is an `int` array `int[]`. This is often called a "jagged array" in comparison with the multi-dimensional array.

As an example, one can declare a 3-dimensional string array variable as follows:

```
string[,,] text;
```

Note the two commas for a 3D array type. One can also initialize the size of the array variable, as is done in the sample code.

For instance, the following declares a variable `truth` of a two-dimensional `bool[,]` type, with 2 "rows" and 5 "columns".

```
var truth = new bool[2, 5];
```

One can even assign the initial values using the array initializer syntax, as we did in the previous lesson. The syntax is slightly different.

```
var box = new[,] {
    {1, 2, 3},
    {4, 5, 6},
};
```

The type of `box` in this case is inferred to be `int[,]`, and its dimension is `2x3`, although they are not explicitly specified.

This statement is equivalent to the following:

```
var box = new int[2,3] {
    {1, 2, 3},
    {4, 5, 6},
};
```

> Note the trailing comma after the last row item in this example. This comma is optional. It is a matter of style, but having the trailing comma (not just between the items) can be a bit more convenient when you add a new item or delete the last item, etc.
>
> Note that we do not use the trailing commas for the column values for a given row in this example.

In the `CreateTable()` method of the sample code, we initialize the variable `table` to a 2D `int` array of `range * range`, and set the values of its elements through nested `for` loops.

```
for (var l = low; l < high; l++) {
        for (var r = low; r < high; r++) {
            // Do something here.
        }
    }
```

When we specify an integer range, it is customary to include the low value but exclude the high value. We follow that convention in this sample code.

As stated, we will not include comments in our code samples in this book, but that information would be a good thing to include in the comments/documentations about the use of the `CreateTable()` and `WriteTable()` methods.

Both `for` loops start from `low` (`l = low`) and iterate to a value one less than `high` (`l < high`). The types of both "loop variables" `l` and `r` are `int` because they are initialized with an `int` value (`low`). The fact that `low` is an `int` is specified in the method signature, `int[,] CreateTable(int low, int high)`.

We could have explicitly declared `l` and `r` as `int`, but that is not really necessary in this case.

The classic `for` loop (`for ( ; ; ) { }`) is based on C's `for` loop. There are three "slots" after the keyword `for` and before the "for block" (`{ … }`). The first slot is typically used for initializing the "loop variables", as in our example. It can be left empty.

The second slot must be a Boolean expression, if present.

As stated, as long as this expression evaluates to `true`, the statements in the `for` block will be repeatedly executed. That is, they keep iterating as long as `l < high`, or `r < high`, for the outer and inner loops, respectively, in this example. If this boolean expression is missing, then it is considered `true`.

The third statement, which is also optional, is executed between iterations. `++` is an increment operator. As in many C-style languages, C# has both prefix and postfix/suffix versions. (The `++` operator is placed before or after the variable, respectively.)

Both versions increment the operand value by 1. That is, `i++`, or `++i`, is equivalent to the following, when used as a statement:

```
i += 1;
```

This is also equivalent to

```
i = i + 1;
```

That is, `i++`, or `++i`, adds `1` to the current value of `i`, and assigns the result back to `i`. Hence, the name "increment operator". (Obviously, we have the corresponding " decrement operators", `--i` and `i--`, as well.)

The values of the expressions `++i` and `i++` are different, however. The value of the expression `++i` is `i+1` whereas that of `i++` is `i`.

For our `for` loops, this distinction is not significant since we are using them as statements (and ignoring their values as expressions). In both cases, the end result is the same. The value of `i` has been incremented by one.

> If this does not make sense to you, do not worry. You are not the only one. In fact, Golang, for example, which is based on C, does not have the prefix versions, for both increment and decrement operators. Apparently, the Go language creators decided that this was *so confusing* to developers that they could not include them in their new language. 😀 Also, the postfix versions in Go are statements, not expressions, unlike in C/C++ or C#.

The name C++ comes from `C` plus `++`. The name C# comes from `C++` plus `++`. Ironically, `C++++` or `(C++)++` is not a valid expression in C#. ☺ We will leave it as something to "think" about for the readers as to why that is the case.

As with other C-style languages like C/C++, Java, or JavaScript, the braces `{ }` after the `for` keyword are optional if there is only one statement in the block. This is true for many statements that use blocks, including `if`, `foreach`, `while`, etc.

The author, however, recommends to always use the braces. This practice can potentially eliminate many possible errors. Remember, the programs change over time.

The `CreateTable()` method computes the multiplication values for two given numbers (line 36), and builds the multiplication table through the two nested loops. The 2D array variable `table` is then used in the `WriteTable` method (line 6).

Note that, as stated, an array is a reference type. When the method return value is assigned to a variable, as in the sample code line 6,

```
var table = CreateTable(low, high);
```

The returned value is "copied". That is, the variable `table` of `CreateTable()` (line 40) is different from the variable `table` of `WriteTable()` (line 6).

They, however, happen to "point to", or reference, the same array, which is created in the `CreateTable()` method. This is because an array is a reference type. We do not have to copy all `range * range` elements of the array, element by element.

In this sample code, although it is a somewhat trivial example, it should be noted that we have divided the function into two methods. One, `CreateTable()`, deals with the "data" whereas the other, `WriteTable()`, primarily handles the "presentation".

The `WriteTable()` method first prints out the "headers".

```
Write("        ");
for (var v = low; v < high; v++) {
    Write($"{v,4}");
}
WriteLine("");
```

In this example, we use 4 spaces for the width of each "table cell", as indicated by the format specifier `{v,4}` in the string interpolation. Note that `Console.Write()` is essentially the same method as `Console.WriteLine()`, but it does not automatically add a newline at the end.

Then, the "divider":

```
Write("       -");
for (var v = low; v < high; v++) {
    Write("  --");
}
WriteLine("");
```

Again, 4 spaces per "cell". (We could have used a `const` for this fixed length.)

How does one know if this prints out what one wants? It’s generally done via "trial and errors". Print out something first and, based the result, change the formatting slightly, etc.

It sounds tedious. But, a lot of programming tasks involve tedious work in case you are new to programming and have romantic ideas. ☺

Finally, the body of the table is printed with "row headers" for each row:

```
for (var l = low; l < high; l++) {
    Write($"{l,4}");                             ①
    Write("   |");                               ②
    for (var r = low; r < high; r++) {           ③
        Write($"{table[l - low, r - low],4}");
    }
    WriteLine("");
}
```

① "Row header".

② Vertical divider.

③ Inner loop where the actual multiplication numbers are printed.

The multiplication table is "2-dimensional". One loop goes over the horizontal axis, or across the columns (the inner loop in this example), and the other loop goes over the vertical axis, or across the rows (the outer loop in this example).

The value `l` corresponds to a number printed on the left hand side. The value `r` corresponds to a number printed on top.

The value of each "cell", at a given row and column, is a product of two numbers, one from the row and the other from the column. These values have been calculated in the `CreateTable()` method.

```
table[l - low, r - low] = l * r;
```

This number is printed with the 4 space width. That's the multiplication table. The nested loops produce the "two-dimensional" printout.

There are a number of different ways to do an iteration in C#. We used the `foreach` statement before. If we use `foreach` instead of `for` in the `WriteTable()` method implementation, for instance, then it may look like this:

```
internal static void WriteTable(int low, int high) {
    var table = CreateTable(low, high);
    var axis = System.Linq.Enumerable.Range(low, high - low);

    Write("        ");
    foreach (var v in axis) {
        Write("{0,4:D}", v);
    }
    WriteLine("");

    Write("       -");
    foreach (var _ in axis) {
        Write("{0,4:S}", "--");
    }
    WriteLine("");

    foreach (var l in axis) {
        Write("{0,4:D}", l);
        Write("{0,4:C}", '|');
        foreach (var r in axis) {
            Write("{0,4:D}", table[l - low, r - low]);
        }
        WriteLine("");
    }
}
```

There is some small differences in these two implementations (for example, we use the `string.Format()` style formatting in this version, just for illustration), but the main difference is that this version uses `foreach` rather than `for`.

In order to do this, we create a variable of a certain collection type, `axis`.

```
var axis = System.Linq.Enumerable.Range(low, high - low);
```

We will learn more about the `System.Linq` namespace later in the book, but the `Enumerable.Range()` static method generates a variable of the interface type `IEnumerable<int>`.

Any variable that implements the `IEnumerable` interface (from the `System.Collections` namespace) or the `IEnumerable<T>` interface (from `System.Collections.Generic`) can be used in the `foreach( … in … )` statement.

The value of `axis` is, roughly speaking, something like this, `{2, 3, 4, 5, 6, 7, 8, 9}`, for `low == 2` and `high == 10`. It is like an array. It can be "iterated". That is how we are able to use `foreach` in this version of `WriteTable()`.

We have not discussed any of these yet, interfaces, generics, LINQ, etc. We have a lot to learn. ☺

Summary

We explored the C#'s multi-dimensional arrays. We also reviewed how to do looping in C# using the keywords `for` and `foreach`. In addition, we introduced the important concept of the "reference type" in this lesson.

The `static class` in C# does not create a new type. Instead, a `static class` is used as a container for the static methods. In C#, we cannot have top level "functions" in a program, unlike in many other C-style programming languages, and hence we use the static methods instead.

7.3. Exercises

1. Write your own program that prints out the multiplication table between 2 and 9, without copying and pasting the sample code in this lesson.

Author’s Note

Why Do You Hate Semicolons So Much? 😀

In some languages like Python and Javascript, the statement-ending semicolons are optional.

In the Python world, virtually nobody uses semicolons. The semicolons are not "Pythonic". In Javascript (which is a C descendent), there are two camps. The people who vehemently object to using semicolons, and the rest who do not care.

For some people, it is a religion. If you use semicolons in some teams, you cannot even check in your code. Through the forced linting, you have to completely strip off the semicolons. Otherwise your code is just "too ugly" to be included in the team’s code repository.

Go is another (opinionated) language that takes this up to another level. Go is based on C, which uses the semicolons. Go’s grammar requires the statement-ending semicolons. And yet, in practice, nobody uses semicolons. You simply can’t. The Go’s creators made sure that nobody could use semicolons in Go programs.

Why? 😀😀😀

Chapter 8. Base 32 Numbers

8.1. Agenda - Type Conversion, `TryParse`, `out` Parameter, Exception Handling

We will go through a number of short code samples, in this and the next few lessons, to cover some fundamentals of C# programming.

> The main focus of the book is to teach the C# programming language. The code samples are primarily chosen for that purpose, and they may, or may not, have any practical values.

8.2. Code Reading - Integer Base Conversion

The following program converts an integer into a string in the "base 32" representation.

base32-numbers/Program.cs

```
using Example;

Console.Write("Input a number: ");
var input = Console.ReadLine();

if (int.TryParse(input, out int num)) {
    var base32Str = Base32.ToBase32(num);
    Console.WriteLine($"In base 32 = {base32Str}");
} else {
    Console.WriteLine($"Invalid number = {input}");
}
```

base32-numbers/Base32.cs

```
namespace Example;

static class Base32 {
    private const int BASE = 32;

    internal static string ToBase32(int num) {
        var (number, flipped) = (num < 0) ? (-num, true) : (num, false);
        return (flipped ? "-" : "") + DivideAndAggregate(number);
    }

    private static string DivideAndAggregate(int n) {
        if (n < BASE) {
            return ConvertIntToChar(n).ToString();
        }

        var (v, r) = (n / BASE, n % BASE);
        var prefix = DivideAndAggregate(v);
        return prefix + ConvertIntToChar(r).ToString();
    }

    private static char ConvertIntToChar(int n) =>
        (n < 10) ? (char)(n + '0') : (char)(n - 10 + 'a');
}
```

8.2.1. Explanation

If we run the program with `dotnet` CLI, it first asks for a number. If a number is supplied, then it prints out the number in base 32.

```
$ dotnet run

Input a number: 12345678   ①
In base 32 = booae         ②
```

① `12345678` is a user input.

② `booae` is the number `12345678` in base 32.

8.2.2. Grammar

We mostly use numbers in base 10. The decimal numbers. But other bases are also commonly used. The most common bases are 12, 24, 60, and, of course, 2, 8, and 16, as they are generally used in computer science.

So, what is the "base"? A base, or radix, is used to represent a number. A number can be represented only using a definite base. As an example, the number `123` written in base 10 means that it is `1 * 100 + 2 * 10 + 3 * 1`.

Likewise, the number `123` written in base 8 means it is `1 * 64 + 2 * 8 + 3 * 1`. The "same" number `123`, in a text representation, has different values depending on what base is used to represent the number.

Because the decimal number representation (base 10) is so pervasive that we often forget that the number representation depends on the base.

In C#, we use special notations. The numbers that start with `0b` are in the binary number representation. Those that start with `0x` are hexadecimal numbers. The numbers that start with `0` followed by another number are octal numbers. Any other number literals in C# use the base 10.

Now, the sample code in this lesson converts a decimal number to a number in base 32. The number 32 is just arbitrary, chosen for illustration.

Since C# does not provide a way to represent numbers in base 32, we will just use strings to represent the numbers in base 32. In particular, we will use the numbers `0` through `9` and then the alphabets `a` through `v` to represent the 32 digits in base 32.

The "main program" accepts a user input, as a string, and it first converts it to an `int`.

There are a number of different ways one can convert a string, e.g., `"100"`, into a number, e.g., `100`, in .NET. The most common way is to use one of the static methods defined in the numeric types.

For example, the sample code uses the `TryParse()` static method of `System.Int32`. The `TryParse()` method returns true if the input is successfully parsed as an `int`. Otherwise it returns false. When it is successful, the parsed integer is returned as an out parameter, `num` in this example.

C# provides a number of keywords to modify the behavior of the method parameters, `in`, `out`, and `ref`. These parameters are passed to the methods "by reference". That is, the change made to the argument inside a method will be visible to the calling context.

To use an `out` parameter, both the method definition and the calling method must explicitly use the `out` keyword.

In the `if` block in the example, suppose that `TryParse()` returns `true`, and the out parameter `num` is set to the parsed integer. Then, we call our base32 conversion method with the given `num`, as we will further discuss shortly.

If the `TryParse()` method returns `false`, then we can assume that the input argument is not a valid number. We print out an error message, in the `else` block, and the program terminates. The actual value of `num` is irrelevant in this case.

Alternatively, one can use the `Parse()` static methods of the numeric types to parse a string into a number. The `Parse()` methods throw an exception if the input string cannot be properly converted to a number of the desired type. They return the parsed number if the conversion is successful.

For example, we could have used `System.Int32.Parse()` instead in our sample code:

```
try {
    var num = int.Parse(input!);
    var base32Str = Base32.ToBase32(num);
    Console.WriteLine($"In base 32 = {base32Str}");
} catch (FormatException ex) {
    Console.WriteLine($"Failed to parse: {ex}");
}
```

Since the `Parse()` method throws an exception, `FormatException` in particular, we use the `try`-`catch` blocks.

If `Parse()` is successful, we use the parsed integer `num` to call `Base32.ToBase32()` to convert the number into the base 32 format. (We can ignore the bang sign `!` after the argument `input` for now.) If the `Parse()` method throws an exception, then we print out the error message (in the catch block).

We will cover the exception handling in C# in more detail in later lessons. But, the basic idea is that if anything happens in the `try` block we can "catch" the errors in one or more `catch` blocks. The `catch` blocks essentially use the "type matching". Based on the types of the exception objects thrown, a particular catch block, and no more than one block, may be selected. We can optionally use a `finally` block, which is executed regardless of whether there is an exception thrown in the `try` block or not.

> There are also other ways to convert a string, or any object that implements the `System.IConvertible` interface, into a number. For example, the `Convert.ToInt32(String)` static method in the `System` namespace converts a given string into an integer. We will discuss interfaces later in the book.

The `Base32` static class includes one `internal` method `ToBase32()`, which takes an `int` and returns its base32 representation as a string. This method is called by the main driver program, as we discussed above.

The `Base32` class also includes a couple of `private` static methods. The `ConvertIntToChar()` is a helper function that converts a number into a `char`, which will be a part of the base32 string representation. This single `char` corresponds to a single digit in our base 32 representation.

We map numbers `0` through `31` to characters `'0'` through `'9'` and `'a'` through `'v'` (32 in total).

Note how it uses the integer addition and then the (char) casting to convert int to the corresponding char in ASCII encoding.

In the low 8 bit space, the Unicode character representation is more or less equivalent to that of ASCII. The important property that this implementation relies on is the fact that the numbers, and the lowercase alphabets, are in consecutive places in the Unicode space, respectively.

More generally, we could have used an explicit mapping from the numbers 0 through 31 to the "base 32 digits", e.g., using IDictionary<int, char>. We will cover the collection types in .NET in later lessons.

Note that we use the "expression body" in defining the ConvertIntToChar() method, using the "fat arrow" (=>) followed by an expression, which incidentally uses the ternary operator which we introduced in the previous lesson.

This is equivalent to

```
private static char ConvertIntToChar(int n) {
    return (n < 10) ? (char)(n + '0') : (char)(n - 10 + 'a');
}
```

Note that return is implicit in the expression-bodied methods. This is also equivalent to the following, using the if statement,

```
private static char ConvertIntToChar(int n) {
    if (n < 10) {
        return (char)(n + '0');
    }
    return (char)(n - 10 + 'a');
}
```

Or,

```
private static char ConvertIntToChar(int n) {
    if (n < 10) {
        return (char)(n + '0');
    } else {
        return (char)(n - 10 + 'a');
    }
}
```

> There is often a tradeoff between brevity and verbosity. Choose the styles that you feel most comfortable with.

In order to compute the base 32 numbers, we use "recursion". In this particular implementation, the method `DivideAndAggregate()` calls itself.

A recursive function needs a base case. In this example, if the given number is less than 32, then we simply return the corresponding "digit" as a string.

```
if (n < BASE) {
    return ConvertIntToChar(n).ToString();
}
```

`ToString()` is a "virtual method" declared in the base type `object` in C#, or the corresponding base class `System.Object` in .NET. In this case, it simply converts the char, returned from `ConvertIntToChar()`, to a single character string.

The recursion relies on the following observation:

```
1234 = 1 x 10^3 + 2 x 10^2 + 3 x 10^1 + 4 x 10^0
```

```
      = ((((1 x 10) + 2) x 10) + 3) x 10 + 4
```

The number `1234` (in base 10) is the same as `(123 * 10) + 4`. Likewise, `123` is the same as `(12 * 10) + 3`. `12` is in turn the same as `(1 * 10) + 2`. finally, `1` is `(0) + 1`.

Therefore, if we know what `1` is in base 10, we can easily compute `12` in base 10. If we know what `12` is in base 10, then we can compute `123` in base 10, using the formula, `(12 * 10) + 3`. And so forth.

With the same reasoning, the number `xyzw` in base 32 is the same as `(xyz * 32) + w`, and `xyz` is `(xy * 32) + z`, etc.

The `DivideAndAggregate()` static method uses this logic to compute the string representation of the given number in base 32.

If we rewrite the method as follows, then it is easier to see. We can easily match `xyzw`, `xyz`, and `w` to the corresponding components in the method definition.

```
private static string DivideAndAggregate(int n) {
    // ...
    return DivideAndAggregate(n / 32) + ConvertIntToChar(n %
32).ToString();
}
```

The internal API `ToBase32()` merely calls the private `DivideAndAggregate()` method other than handling the minus sign for negative numbers.

This method can also be written in a more verbose way, if that enhances the readability for you.

```
internal static string ToBase32(int num) {
    var flipped = false;

    if (num < 0) {
        num *= -1;
        flipped = true;
    }

    var base32Str = DivideAndAggregate(number);

    if (flipped) {
        base32Str = "-" + base32Str;
    }

    return base32Str;
}
```

Summary

We looked at a few different ways to convert a string into a number. In particular, `Parse()` and `TryParse()` static methods defined for all numeric types, integral or floating point, as well as for the Boolean type, can be used for type conversion.

We also briefly looked at the exception handling in C# in this lesson.

8.3. Exercises

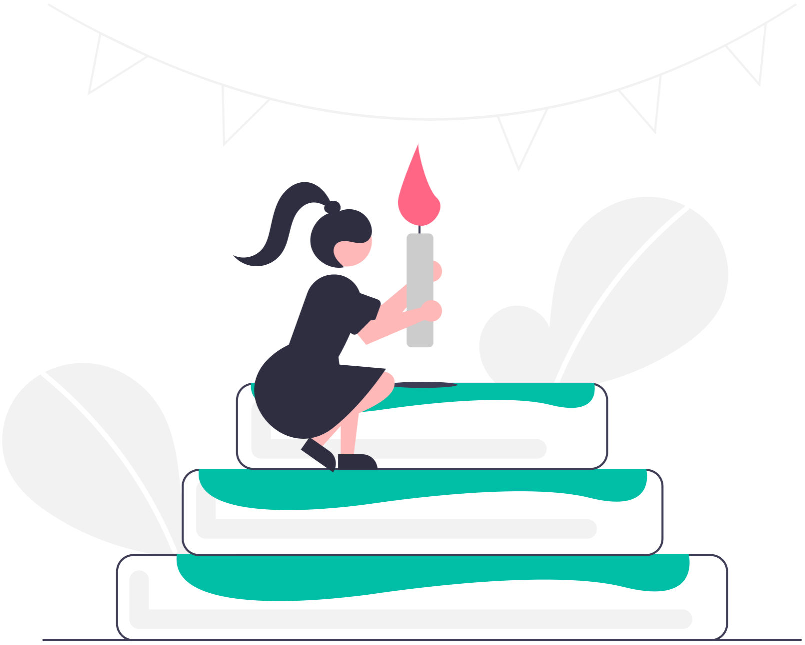
1. Write a static method that converts a given (decimal) integer into a binary number. Write a program that converts a decimal integer number, given as a command line argument, to a binary number representation and prints out the result.

Chapter 9. Birth Date

9.1. Agenda - Enum types, `while` and `do while` Loops, `switch` Statements, `char` Type

Handling date and time is one of the most common tasks in programming. We will take a look at some simple APIs in the .NET `DateTime` class (from the `System` namespace).

We will also review the character and string types in C#.



9.2. Code Reading - Date and Time

How old are you *in days? In hours and minutes?* ☺ What day of the week was it when you were born?

The following program does simple math for you to answer these questions. The main logic of the program is all in the main "Program.cs" file:

birth-date/Program.cs

```
using Example;

var smiley = char.ConvertFromUtf32(0x1F642);

while (true) {
    Console.WriteLine("What is your birth date? (mm/dd/yyyy)");
    var input = Console.ReadLine();

    if (DateTime.TryParse(input, out DateTime date)) {
        var weekday = BirthDate.Weekday(date);
        Console.WriteLine($"You are a {weekday} child.");

        var days = BirthDate.AgeInDays(date);
        Console.WriteLine($"You are {days} days old as of today.");

        var (hours, mins) = BirthDate.AgeInHours(date);
        Console.WriteLine($@"You are {hours} hours
    and {mins} minutes old as of right now. {smiley}");

        break;
    } else {
        Console.WriteLine("Invalid date. Try again? Yes (Y) or No (N)");

        var ans = Console.ReadLine() ?? "";
        var firstLetter = (ans.Length > 0) ? ans.Substring(0, 1).ToUpper() : "";
        switch (firstLetter) {
            case "Y":
                continue;
            case "N":
            default:
                Console.WriteLine($"Your loss! {smiley}");
                return;
        }
    }
}
```

The `Birthdate` static class *internally* exports a few simple wrapper methods around some `System.DateTime` APIs. (Note: "Internal" within the assembly.)

birth-date/Birthdate.cs

```
namespace Example;

static class BirthDate {
    internal static string Weekday(DateTime date) => Enum.GetName(date.DayOfWeek) ?? "";
    internal static int AgeInDays(DateTime date) => (DateTime.Now - date).Days;
    internal static (int, int) AgeInHours(DateTime date) {
        var age = DateTime.Now - date;
        return ((int)age.TotalHours, age.Minutes);
    }
}
```

9.2.1. Explanation

If we run the program with the `dotnet` CLI tool, it first asks for a date. If a date is provided, then the program prints out some (trivia) information on the console.

```
$ dotnet run

What is your birth date? (mm/dd/yyyy)
1/1/2000                                      ①
You are a Saturday child.
You are 7846 days old as of today.
You are 188312 hours
    and 50 minutes old as of right now.
```

① User input.

Here’s another sample run:

```
$ dotnet run

What is your birth date? (mm/dd/yyyy)
january 50th 30000                            ①
Invalid date. Try again? Yes (Y) or No (N)
yes                                           ②
What is your birth date? (mm/dd/yyyy)
jan 1 2020                                    ③
You are a Wednesday child.
You are 541 days old as of today.
You are 12993 hours
    and 35 minutes old as of right now.
```

① An invalid user input.

② The user types "yes" and presses Enter.

③ A new valid date input.

9.3. Grammar

C# and .NET uses Unicode characters for `chars` and `strings`. Unicode uses 2 or 4 bytes to represent all characters in all the (known) languages in the (known) universe. ☺

We use certain encoding schemes to represent Unicode characters on computer (e.g., for storage and transmission). Two of the most popular encoding methods for Unicode are UTF-8 and UTF-16. The English alphabets, and most of the Latin alphabets, require only 1 byte. And, UTF-8 is the most popular, and fairly efficient, encoding method when transmitting or storing the data in Unicode characters.

Without going into too much detail, C# internally uses one or two values of the `char` type (of 2 bytes) to store a single Unicode character, using UTF-16. In many cases, a single `char` in C# corresponds to a single Unicode character (e.g., from the "Basic Multilingual Plane" (BMP)). In some cases, however, a single `char` is not enough for certain Unicode characters (e.g., a character that requires an additional 2 bytes from a "Surrogate Plane").

In this sample code, we use the function `char.ConvertFromUtf32()` to get a certain smiley character (`\u1F642`). This static method from `System.Char` in .NET returns a `string`, not a `char`, because some Unicode characters cannot be represented by a single `char`.

Normally, any Unicode character from the BMP (that requires only 2 bytes) can be represented by using the "\u" character escape syntax in C#.

For example, `\u0041` represents a character `'A'`. (A hexadecimal number `0x41` is `65` in base 10. Note that we use *four* hexadecimal digits for two bytes in this notation regardless of the integer value of the `char`.)

For any other characters, we will need to rely on some API methods provided by .NET. Note that the `string? ConvertFromUtf32(int)` method takes an `int` argument, not a `char`.

In this particular example, the integer `0x1F642` does not fit into a char type. An explicit casting to `char` would have been a compile time error.

> We do not use the "unchecked", or unmanaged or unsafe, features of C# in this book.

C# provides a number of different constructs for "looping", or repetitions.

The `while` statement executes a statement or a block of statements while the specified Boolean expression (in the parentheses right after the keyword `while`) evaluates to `true`. Because that expression is evaluated before each execution of the loop, a while loop executes zero or more times.

Here's an example of the `while` statement:

```
var i = 0;
while (i++ < 10) {
    Console.WriteLine("Hello again!");
}
```

> How many times do you think it will print out the `Hello again!` sentence? We discussed how the increment operator expressions are evaluated (prefix vs postfix versions) in the earlier lessons. If you are confused, do not worry. Some people hated the increment/decrement operators so much that they butchered them into oblivion (in certain programming languages). 😃 If you have done some programming in Go or Rust, then you probably know what we are talking about here. ☺

On the other hand, a `do {} while()` loop executes the block/statement at least once regardless of the value of the Boolean expression. Here's an example of the `do while` statement:

```
do {
    Console.WriteLine("Hello at least once!");
} while (false);
```

This prints out `Hello at least once!` to the console, once, even if the while Boolean expression is always `false`. Notice the trailing semicolon at the end of the `do` statement. (There is, however, no semicolon after the `while` block.)

The `while` statement in the sample code is an "infinite loop". The Boolean expression is always `true`. In order to exit out of a loop (infinite or not), before its normal completion, we use the C# keyword `break`.

Note that the loops can be nested (e.g., a `foreach` loop inside a `while` loop, etc.) and the `break` statement only breaks out of the innermost loop, or the `switch` statement, that encloses the statement.

Inside the `while` loop in the sample code, we have an `if-else` statement. If the user input is in a valid date format, then we perform our main task. Otherwise, we ask the user if he/she wants to try again.

The `else` block uses a `switch` statement. `switch` is a selection statement that chooses a single switch section, or a `case`, to execute from a list of candidates based on a pattern match with the match expression. The `switch` statement is often used as an alternative to an if-else construct if a single expression is tested against two or more conditions. In this particular example, we could have simply used an `if-else` statement.

The match expression (in the parentheses) can be any valid expression, in the "modern C#", which evaluates to a (compile-time) value of *any type*.

In the sample code, the type of `firstLetter` is a string. And its value is compared with `"Y"`. If it is `true`, then it "continues", by going back to the beginning of the `while` loop. If not, then the `default` case is executed. It merely `returns` in this case, thereby effectively terminating the program.

Note that the test for `"N"` is redundant in this example. It simply illustrates that one can have multiple cases for matches. Without an explicit `break` (or, `return`), the cases "fall through".

Now if the user inputs a valid date string, then we convert it into a `DateTime` value using the `TryParse()` method. This `DateTime.TryParse()` method also uses the `out` parameter to return the parsed datetime when it returns `true`.

`DateTime` is a value type, and hence we need the `out` modifier to get the value from inside the method.

> The `ref` keyword indicates that a value is "passed by reference", as an argument to a method or as a return value.
>
> The `out` keyword is like `ref` except that it can only be used for a method argument. The `out` parameter need not be initialized before it is passed in to a method unlike `ref`. The `in` parameter behaves like `ref` parameter but the value cannot be modified.
>
> We will discuss "value" vs "reference" in more depth in Part II, and throughout this book.

Then it uses the convenience static methods that we have defined in the `Birthdate` static class to produce the desired output.

A value of `DateTime` has an "instance property" `DayOfWeek`, which returns the day of the week for the given object as a value of an enum type, `DayOfWeek`. The `DayOfWeek` type defines a set of integer values from 0 to 6 to represent the days from Sunday to Saturday.

The `Enum.GetName()` function returns the "string" value for the given enum. That is, `"Sunday"`` for `DayOfWeek.Sunday` (which is defined to be an int `0`), etc.

Note that `Enum.GetName(date.DayOfWeek)` is a "generic method", as we will discuss in the coming lessons. This is equivalent to `Enum.GetName<DayOfWeek>(date.DayOfWeek)`. The type parameter `DayOfWeek` (in `<` and `>`) can be omitted since the compiler can infer the type from the argument `date.DayOfWeek`, which is a value of type `DayOfWeek`.

The `??` operator is known as the "null-coalescing operator" in C#. It is a binary operator.

If the value of the first argument (before `??`) is `null`, then the value of the second argument is used as the value of the whole expression, e.g., `Enum.GetName(date.DayOfWeek) ?? ""` in this example, which is an empty string `""`. Otherwise, the value of the expression is that of the first argument, that is, `Enum.GetName(date.DayOfWeek)` in this example.

In effect, the result of this expression cannot be `null` (unless the expression of the second argument can result in null, which beats the purpose of using the null-coalescing operator, however).

We declare that the static `string Weekday(DateTime)` method returns a non-nullable `string` value (not a nullable string `string?`), and we deliver on that promise.

The methods, `int AgeInDays(DateTime)` and `int AgeInHours(DateTime)`, are implemented in a similar way.

The value of the expression `(DateTime.Now - date)` is the difference between "now" and the given time, and it evaluates to a value of type `System.TimeSpan` (not `System.DateTime`). It is a "readonly" value type. That is, the value cannot be modified once it is created.

`Now` is a "static property" defined on the `DateTime` readonly struct, which returns the current time as a `DateTime` value.

Note the versatility of the C# syntax. The difference between two `DateTime` values, using the `-` operator, is a value of type `TimeSpan`. We will discuss further on the topic of "operator overloading" later in the book.

`Days`, `TotalHours`, and `Minutes` are the properties, or the instance properties, of the `TimeSpan` struct. Note the "dot syntax". For both (static and instance) methods and properties, we use the dot syntax. But the entities that precede the dots are different.

Properties are just like methods, but they are primarily used to access the data encapsulated in an object.

Note that the main program calls all three methods in a similar fashion, with the same argument.

We could have implemented this more effectively, for example, by using a composite type like tuples. (`AgeInHours()` does use tuples.) Or, we could have defined a custom type to make this program "better", or more easily readable, or more easily maintainable, etc.

We will learn how to define and use the custom types, using `struct`, `class`, and `record`, starting from Part II.

One last thing to note in the sample code of this lesson is the use of `@`. The `@` symbol can be used to "escape" the identifiers/names in a C# program. For example, `for` is a reserved language keyword, and it cannot be used in a program as an identifier like a variable name.

C#, however, allows the use of those reserved words through the `@` escaping. That is, `@for` is a valid C# identifier/name. It is `"for"`, not `"@for"`, that is used in the compiled program, but the compiler knows that this `for` (as indicated by the preceding `@`) is not the C# keyword `for`. (If you have used Python before, the trailing underscore {_} in a name in the Python programs plays the similar role.)

Another use of the `@` symbol is for the "raw strings", or the verbatim string literals, as introduced before.

In a string with a form, `@"…"`, with a prefix `@`, special symbols (like a tab or a newline) do not have to be escaped. In fact, they cannot be escaped. All characters (or, most of them, more precisely) in a raw string are interpreted as is (as a "raw" character). There are a few exceptions, however. For example, a double quote character (which is used to enclose the raw string literal) should still be escaped as `""` (two double quotes).

The raw string and the `$`-interpolated string formats can be combined. For instance, in `$@"…"`, most characters need not be escaped, and we can still use the `{}` interpolation. In the `$`-string interpolation, including the raw interpolated string literals, the curly braces have to be escaped. For `{` and `}`, we need to use `{{` and `}}`, respectively.

Note, in the sample code, that the newline as well as a few spaces, between "hours" and "and", are printed in the console output.

Summary

We reviewed the `while` and `do while` loop constructs in this lesson as well as `switch` statements.

We introduced the `enum` types, which are user-defined integer types. We will show more example uses of the enum types throughout this book.

The `char` (Unicode) and `DateTime`, as well as `TimeSpan`, are a few of the most commonly used types in C# programs.

9.4. Exercises

1. Validate the input date. For example, the "birth date" cannot be later than "now". Write the same, or similar, program to this lesson’s sample program (without copying it), but add any validation logic which you see to be necessary. How would you handle the invalid input?
2. Write a program that, given a birth year, month, and day, prints out the "age in seconds".
3. In the previous lesson, Base 32 Numbers, we wrote a program that converts an integer with base 10 to a number representation in a different base, e.g., 32. We used the "recursion" in its implementation. Now that we learned the `while` loop, we can do it "iteratively" as well, that is, using a loop instead of recursion. This can be a bit difficult problem depending on your prior programming experience, but give it a try! ☺

Author’s Note

What It Takes to Be a Good Programmer

Programming, or more broadly software development, involves a lot of different skills and talents.

First, you will need to be an expert in the language you use in programming. This is actually the easiest skill that you can learn. The programming languages have well-defined grammar, unlike spoken languages, with a finite set of rules. The resources like this book can help you learn the languages.

Second, you will need to be familiar with commonly used libraries and APIs. It takes experience. The longer you program, the more familiar you will become with various libraries. It is not "difficult", but it just takes time.

Third, you will need some analytical or problem solving skills. This is not something you can learn by reading books or anything like that. But, you will get better over time as you train yourself by working on more problems. It helps to learn some common algorithms as well. The "familiarity" with a diverse set of problems will be definitely helpful when you face a new problem.

And, eventually, you will need to develop the system or architectural design skills. This is different from the programming skills. Creating a large software is like "building a pyramid", using the analogy that we used before, or building a "starship" using Lego blocks. You can only learn this skill by actually doing it, that is, by building a large scale software.

Chapter 10. MD5 Hash

10.1. Agenda - Extension Methods, `using` Statement, `IDisposable`, `StringBuilder`

We will review some more simple C# programs. We will introduce another use of the `using` keyword in this lesson, among other things.

10.2. Code Reading - Content Hashing Using MD5

The sample code demonstrates the use of the `MD5` hashing method.

crypto-md5/Program.cs

```
using Example;

var input1 = "hello world.";
var hashed1 = CryptoMD5.Hash(input1);

Console.WriteLine($"{nameof(input1)} = {input1}");
Console.WriteLine($"{nameof(hashed1)} = {hashed1}");

var input2 = "hello world!";
var hashed2 = CryptoMD5.Hash(input2);

Console.WriteLine($"{nameof(input2)} = {input2}");
Console.WriteLine($"{nameof(hashed2)} = {hashed2}");

var input3 = "Hello World!";
var hashed3 = CryptoMD5.Hash(input3);

Console.WriteLine($"{nameof(input3)} = {input3}");
Console.WriteLine($"{nameof(hashed3)} = {hashed3}");
```

The `CryptoMD5` static class includes a simple convenience method, `string Hash(string)`, which is declared as `public` in this example.

crypto-md5/Crypto.cs

```
namespace Example;
using System.Text;
using System.Security.Cryptography;
using Helper;

public static class CryptoMD5 {
    public static string Hash(string input) {
        using var md5 = MD5.Create();
        var inputBytes = Encoding.ASCII.GetBytes(input);
        var hashBytes = md5.ComputeHash(inputBytes);

        return hashBytes.ToHex();
    }
}
```

Another static helper method, `string ToHex(this byte[])`.

crypto-md5/Helper.cs

```
namespace Helper;
using System.Text;

static class Extensions {
    internal static string ToHex(this byte[] bytes) {
        var sb = new StringBuilder();
        for (var i = 0; i < bytes.Length; i++) {
            sb.Append(bytes[i].ToString("X2"));
        }
        return sb.ToString();
    }
}
```

This method has the `internal` access modifier in this example.

10.2.1. Explanation

If you run the program, e.g., using the dotnet CLI tool,

```
dotnet run
```

You get the following output:

```
input1 = hello world.
```

```
hashed1 = 3C4292AE95BE58E0C58E4E5511F09647
input2 = hello world!
hashed2 = FC3FF98E8C6A0D3087D515C0473F8677
input3 = Hello World!
hashed3 = ED076287532E86365E841E92BFC50D8C
```

10.2.2. Grammar

"Hashing", broadly speaking, is a one way function. Given a value or data, you can compute its hash value. But, given its hash value, you cannot reconstruct the original data.

Hashing is used in many applications. One of the simplest use cases is to use it as a data "checksum" to verify the integrity of the data.

Without having to compare every byte in a given data set (e.g., file content), we can just compare the hash values, e.g., before and after the transmission over the Internet, and check whether the data has been corrupted or otherwise tempered with.

The "MD5" is one of the simplest and most commonly used hashing methods. .NET includes a support for MD5 as well as for many other algorithms. For example, refer to the dotnet doc, MD5 Class [https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.md5], for more information.

One interesting thing about the type `MD5` is that an instance of `MD5` needs to be "disposed of" after use. Many functions and methods use system resources (memory being the most common such resources) and they need to be cleaned up (e.g., while the program is running).

> ℹ .NET is a managed platform. .NET, in particular, handles the memory management on behalf of a running program. It periodically checks the memory usage and it cleans up any "no-

> longer used" memories. This is known as the "garbage collection".
>
> Although the C# programming language itself does not have such a concept, it does not have any support for the memory management (outside the "unsafe" use), and therefore C# programs can only run on the managed platforms like .NET.

The main program calls the `CryptoMD5.Hash()` static method three times with three different input strings. It just prints out the input strings and their hashed values (as text) for visual inspection, and it terminates.

Note the use of the `nameof()` operator. It returns the string representation of a name used in the program.

As stated, most names used in a program are "dummy" names. That is, changing the names (as long as they are done consistently across the entire program) does not affect the workings of the program. We could have just as well used a name `firstUserInput` instead of `input1`, for instance.

A minor problem occurs, however, when you change the names in most part of the program, but not in the string literals. For instance, in this sample code, we print out the text, `input1 = hello world.`. The string `input1` is tied to the variable name in the program. But, the compiler cannot know this relationship. If one changes the variable name `input1` to `text1` across all its uses in a program, then the program will still work. The only place we can notice that something happened is the string output. In the output, we may still print out `input1 = hello world.` although we changed its variable name to `text1`.

The `nameof()` expression can be used to get the name of a variable or a constant, as well as the name of a member in a type, in a program. This is mainly useful for the diagnostics purposes, etc.

Note that the argument of `nameof()` is a name/identifier and not a string. We mentioned the `@`-escape syntax in the previous lesson. What would be the value of the expression `nameof(@while)` if we declared a variable named `@while`?

The `Example` namespace contains one public static class `CryptoMD5`, which in turn contains one public static method, `Hash()`. The `Hash()` method uses the .NET `MD5` class defined in the `System.Security.Cryptography` namespace.

The `MD5` type "implements" the `System.IDisposable` interface, and it implements the `void Dispose()` method of that interface.

The implementation of the `IDisposable` interface is an indication, e.g., by the creator of the type, that an object of the type uses the system resources and they need to be cleaned up by calling the `Dispose()` method. Not doing so will (likely) lead to a resource leak.

One of the common ways to call `Dispose()` is from inside the `finally` block. We will cover the exception handling in C#, and `try/catch/finally` in particular, later in the book. But, for now, there is an easier way.

C# supports an automatic cleanup of the `IDisposable` resources using the `using` statement. For an object created with `using`, the `Dispose()` method does not have to be called explicitly. At the end of the enclosing block (e.g., the method block), the resources are automatically released.

> Behind the scene, the C# compiler creates a `try/finally` block and generates a code to call `Dispose()` inside `finally`.

In the code sample, the `md5` object is created via the `using` statement:

```
using var md5 = MD5.Create();
```

This is equivalent to

```
using MD5 md5 = MD5.Create();
```

Note that `MD5` uses a factory method to create an instance instead of using the constructors or initializers (as we will see in the second part of the book).

The scope of the variable `md5`, created this way, is until the end of the method body in this example. Hence, when the method returns, the resources that `md5` has used will be automatically cleaned up (e.g., by calling `Dispose()` from the generated code).

Another way to use `using` is to use it as a block statement. For instance,

```
using (var md5 = MD5.Create())
{
    // ...
}
```

In this case, the scope of the `md5` object is that block, and `md5` is cleaned up at the end of the block.

Generally, the single-line statement syntax is more convenient to use, and it is preferred, unless there is a special need to make the scope narrower. In this example, if we use the `using` block, it may look like this:

```
static string Hash(string input) {
    byte[] hashBytes;
    using (var md5 = MD5.Create()) {
        var inputBytes = Encoding.ASCII.GetBytes(input);
        hashBytes = md5.ComputeHash(inputBytes);

    }
    return hashBytes.ToHex();
}
```

💡 Note that any variable of a System.IDisposable type of .NET, or of a type that implements IDisposable, including any user-defined types, can be used in the C# using statement. We have seen similar connections before, for example, between the C# foreach statement and the collections that implement System.IEnumerable or System.IEnumerable<T>.

ℹ The keyword using has many uses. One other use of using is to define an "alias" for a type. This has, however, rather limited uses, as of C# 9.0. But, with the global using feature coming in version 10.0, it can become very useful.

The MD5.ComputeHash() instance method takes a byte array byte[] as an input and returns a byte array as an output.

We convert the input string into byte[] using the Encoding.ASCII.GetBytes() method.

We convert the output byte array into a string using a helper method ToHex(). Note the syntax. There is this this keyword before the argument list in the method declaration. This is called an extension method.

```
static string ToHex(this byte[] bytes) {
    // ...
}
```

The extension methods are no different from any other static methods. But, syntactically, one can use the "dot notation" syntax as if the method is a part of the given type. That is, one can use `hashBytes.ToHex()` rather than `Extensions.ToHex(hashBytes)`, in this example.

The extension methods can only be defined within static classes, and the name `Extensions` in this sample code is arbitrary. We could have included `ToHex()` in the `CryptoMD5` static class as well.

The `ToHex()` extension method converts a given argument of type `byte[]` into a `string` by first converting each byte to a hexadecimal number.

Note the use of `StringBuilder` in its implementation. Repeated uses of the string concatenations can affect performance. It is generally preferred to use a `StringBuilder` object to "build" a string value from the smaller components (e.g., `strings` or `chars`).

The `StringBuilder` type includes many overloaded versions of the `Append()` method. In this example, the argument of the `Append()` method is a (one-character) string throughout the iterations. The `StringBuilder.ToString()` method finally converts the internal data to a string output.

The main program, which is essentially a test driver for `CryptoMD5.Hash()`, calls this method three times. Each time, the `Cryptography.MD5` object is created and destroyed. This may not be the most desirable implementation depending on the particular requirements.

One way to make the program more efficient is to reuse the `MD5` object once created, e.g., throughout the lifetime of the program. This can be done, for example, by creating a custom type, which we defer until the second part of this book.

One thing to note from the output of this sample code is that the hash values are very different even for the rather similar input strings. For instance, for `"hello world."` and `"hello world!"`, we get two completely different hash values.

```
input1 = hello world.
hashed1 = 3C4292AE95BE58E0C58E4E5511F09647
```

```
input = hello world!
hashed = FC3FF98E8C6A0D3087D515C0473F8677
```

For an input string, `"hello world"`, its MD5 hash value is `5EB63BBBE01EEED093CB22BB8F5ACDC3`, another completely different value.

This is a characteristic of a good hashing function. One cannot easily guess an arbitrary input based on the hash values of any known similar inputs.

Summary

We demonstrated, in this lesson, the use of some simple APIs from the `System.Text` and `System.Security.Cryptography` namespaces in .NET.

We introduced the extension methods in C#. We also showed another use of the `using` keyword in the context of the `IDisposable` type variables.

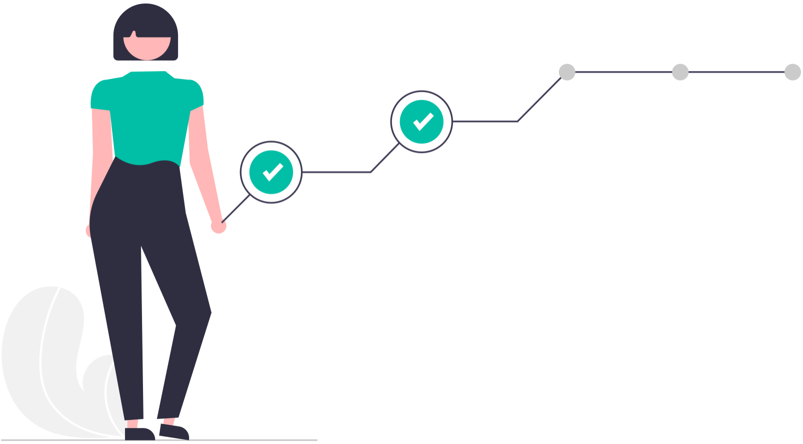
10.3. Exercises

1. Write a program that accepts a string as a command line argument and produces the MD5 hash for the given string.
2. Write a similar program that produces the `SHA256` hash value. Here's a relevant doc: SHA256 Class [https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.sha256?view=net-5.0].

Chapter 11. Find the Biggest Number

11.1. Agenda - Generics, Type Constraints, Generic List, List Initializer, `default` Operator

We cover some more basics of C# programming in this lesson. In particular, we will introduce the collection types and generics, among other things.



11.2. Code Reading - `TryFindMax` Function

This is a program that finds the maximum value from a given list of integers. Here's the main driver.

find-largest/Program.cs

```
using Example;

var list = new List<int>(){
    17, 9, 21, 33, 22,
};
Console.WriteLine($"Input list: {{{string.Join(", ", list)}}}");

if (Finder.TryFindMax(list, out int max)) {
    Console.WriteLine($"Max = {max}");
} else {
    Console.WriteLine("No max found.");
}
```

The core logic is implemented in the static class, `Finder`.

find-largest/Finder.cs

```
namespace Example;

static class Finder {
    internal static bool TryFindMax<T>(IList<T> list, out T? max) where T : IComparable<T> {
        if (list.Count == 0) {
            max = default;
            return false;
        }

        max = list[0];
        foreach (var v in list) {
            if (v.CompareTo(max) > 0) {
                max = v;
            }
        }

        return true;
    }
}
```

The `TryFindMax<T>()` method may look a bit strange, if you have not seen any generic methods before. But, no worries. We will cover the basics of generics in this lesson.

11.2.1. Explanation

If we run the program, e.g., via `dotnet run`, then it prints out the input list and the biggest value among the items in the list.

```
$ dotnet run

Input list: {17, 9, 21, 33, 22}
Max = 33
```

In fact, you can verify that it has found the correct maximum value in this particular example, e.g., just by visually inspecting the example input and output.

> We will briefly discuss automatic testing later in the book.
>
> There seems to be a bit of confusion regarding software testing in the community. Some people ask whether all software need to be tested.
>
> The software testing can be classified in many different ways, including unit testing vs integration or system testing, black-box testing vs white-box testing, and manual testing vs automatic

testing, etc.

The question is not *whether*. The answer to this question is always *yes*. All software *must* be tested. In some way. To some degree.

The real question is *how* and *how much*. Some developers religiously follow the TDD (test driven development), in which they write automatic test programs first *before* they even start working on the main software. Some developers insist on the "100% code coverage", which means that every branch of the code execution flows (e.g., `if` vs `else`) needs to be tested.

The author, personally, does not believe in any of these extremes. In many cases, what is "reasonable" dictates what kind of testing, and to what degree, needs to be done. Sometimes "good enough" is good enough.

11.2.2. Grammar

Generics is a very important part of the C# programming language.

It is not just an "advanced topic" which the beginning programmers can ignore "until later". You may not need to create your own generic classes or generic methods (in the beginning), but it is nonetheless essential to understand the basic concepts of generics.

This is similar to the use of `delegates`. You may not have a need to define your own delegates "until later", but you will still need to understand what they are and how to use them. Note that we introduced the delegates rather early in this book.

The main program calls the generic `TryFindMax()` method with an "integer list" (`List<int>`).

.NET defines a number of "collection types". It supports a list, which is a dynamically sized list data structure, and a dictionary, which is a key value mapping data structure, among other types.

One thing to note is that there are two different kinds for each collection type and their related types such as interfaces, etc.

One kind is a "non-generic" kind. It stores any object, value or reference, as a base type `object`. Since all types in C# inherits from `object` (`System.Object` in .NET), we can store any objects in a non-generic collection instance. These are there primarily for historical reasons, however.

If you use non-generic collections (e.g., from the `System.Collections` namespace), you may end up losing most, if not all, of the benefits of using the strongly typed languages like C#. We will not use any non-generic collections in this book.

The other kind is `generic` collections. The generic collection types can support a broad range of specific types as their elements, like `int` or `string`, not just the broadest possible type `object`.

The catch is that when you instantiate a generic collection type, you need to specify a specific type. That is, although a generic type or generic method, in general, can support a range of types, e.g., more than one types, for its "associated type(s)", one can only use the generic type/method by fixing its associated type(s) to a specific type(s) (allowed by the generic definitions).

The variables of an instantiated/specialized generic type are no longer "generic". They only work with that specific type.

The sample code uses a generic list `System.Collections.Generic.List<T>`, which is sort of a dynamic array, whose size can change at run time unlike the builtin array type.

The T in the angular brackets (<…>) is known as the "type parameter". It is just a placeholder (for the associated type), and it is customary to use the letter T (T for "Type"), if there is only one type parameter, although that is not required.

As stated, we define a generic type (and, a generic collection) "generically" (that is, over a range of different types), but we can only use it for a particular fixed associated type (i.e., its element type).

In the example, we use the int type:

```
var list = new List<int>();
```

This statement declares a variable list of type List<int>. An instance of the List<int> type can contain elements of type int, but nothing else. Note the relationship between the collection type List<int> and its associated element type int.

> If you are new to C# and haven't used any generics before, then the syntax using the angular brackets might seem strange at first.
>
> But, List<int> is conceptually no different from, for example, ListOfInts. Unlike the arbitrary names ListOfInts or List_Of_Integers, etc., however, the compiler can do a lot for List<int>. For one thing, you cannot store objects of any types other than int into List<int> (because that is the contract imposed by the generics).

A list (generic or non-generic) can be initialized using the list initializer syntax (or, more broadly, the "collection initializer" syntax). In the example, we use a pair of curly braces to specify a set of initial integer values, just like the way we do with the array initializations.

The type of `list` declared this way (using `var`) is `List<int>?`, that is a nullable list of `int`s. This fact does not concern us for this particular example. One could have explicitly declared the variable as the type `List<int>`.

Futhermore, note that it is often customary to use an interface type (which is considered "broader") than an implementation type (which is considered "narrower"). In the case of the generic class `List<T>`, it "implements" a generic interface `IList<T>`. Therefore, `List<int>` implements `IList<int>`.

We could have declared `list` as follows, since we do not use any implementation details specific to the `List<int>` implementation.

```
IList<int> list = new List<int>(){
    17, 9, 21, 33, 22,
};
```

Or, we could have just declared it as `List<int>`. In this small example, there is little difference.

If we declare a variable with a specific implementation type, then we can use the shorthand `new()` syntax on the right hand side because the type of the right hand side expression is fixed by the explicit declaration of type `List<int>`.

```
List<int> list = new() {
    17, 9, 21, 33, 22,
};
```

We will go over the basic concepts of the "object oriented programming" (OOP) later in the book, in particular, in the first few lessons of the next part. For now, if anything does not make sense to you, then you can just ignore them and move on.

> As a general comment, you will get most out of this book by reading through the book from beginning to end (or, as far as you can go) rather than trying to understand every detail.

The main program then calls the `TryFindMax()` static method of the `Finder` class with this test integer list.

The `TryFindMax()` method is defined using generics. Although we only use it for `List<int>` in this particular case, there can be benefits for working on a slightly more general, or broader, problem. In any case, this is for illustration.

Let’s look at the method declaration:

```
bool TryFindMax<T>(IList<T> list, out T? max) where T :
IComparable<T> { }
```

We discussed the `internal` access modifier, which makes this method accessible by anyone in the same assembly, and the `static` modifier, which makes it behave like a function, not an object member method.

The method returns a `bool` value, and its name has a type parameter appended at the end. This indicates that `TryFindMax<T>()` is a generic method. The associated type parameter, `T` by convention, can be used, once declared this way, in the method parameter list as well as in the body of the method.

In general, `T` can be any type. In certain method definitions, or type definitions, however, using an arbitrary type may not make sense. In this particular example, the method is supposed to return the biggest value among the elements in a given list.

If a particular type `T` has no concept of "biggest", or "bigger" more generally, then this method will not make sense for such a type. This is where the `where` clause comes in. It is called the "type constraint".

We are declaring that we are defining the `TryFindMax<T>` method only for the types that implements the interface `IComparable<T>` from the `System` namespace, not for any type. (Note that we are using `using System` in virtually every source file when we use anything from the `System` namespace.)

That is, one cannot use our `TryFindMax<T>` method with an associated type that does not implement `System.IComparable<T>`.

`System.Int32` (or, `int`), as well as all other numeric types in C#, happens to implement `System.IComparable<System.Int32>` (or, the corresponding `IComparable<T>` specializations). That is, the values of the `int` type are "comparable". (We will discuss the interfaces in more detail later in the book.)

Why is this important? This is because the implementation uses the fact that an object of that type has a method `CompareTo(T)` (as defined in `IComparable<T>`). This is a "codification" of the condition, as stated, that the values of the type need to be "comparable". Otherwise, the "max" cannot be defined.

Let's first take a look at an example which does not use generics. Let's keep using `int` for simplicity.

```
internal static bool TryFindMaxInt(IList<int> list, out int? max) {
    if (list.Count == 0) {
        max = 0;
        return false;
    }

    max = list[0];
    foreach (var v in list) {
        if (v > max) {
            max = v;
        }
    }
```

```
        return true;
    }
```

This is essentially a "specialization" of the generic version, `TryFindMax<T>()`, for `int`. (Note that there is no more `<T>` in this definition.)

In fact, when we build a program that includes any generic types or generic methods, the compiler generates a number of different versions of those types/methods with associated types as used by the program.

In our sample code, we use the method `TryFindMax<T>()` for `T is int`, but for nothing else. Hence, the compiler generates only a version of the `TryFindMax<T>()` method with `T` being `int`.

ℹ It is worth mentioning that generics supports the type-based "polymorphism". Although we do not commonly use the term polymorphism in this context (in the OOP), it is indeed a polymorphism (which literally means that a thing can have multiple forms). It is often called the "parameterized polymorphism".

Generics is really a concept borrowed from the functional programming. In the pure functional languages like Haskell with static typing, for instance, the "types" are the most essential components of the language (along with the "functions").

Although generics has been used for a long time in the imperative programming languages (like C++), the modern C# uses a lot of functional programming features, besides generics, as alluded in the prelude of this book. In fact, most, if not all, of the modern features in C# described in this book have some functional flavors (although we will not always mention them explicitly).

The method `TryFindMaxInt()`, as well as its generic counter part `TryFindMax<T>()`, uses the "TryXXX method convention", as we discussed in the earlier lessons.

Instead of throwing an exception, the "Try" method returns a `false` Boolean value when it runs into an unusual situation. When it runs successfully, it return a `true` value, and the result of the operation is returned as an `out` parameter, `max` in this example.

When an empty list is passed, there is no "max" value, and hence we return `false`. Note that, even in this case, the `out` parameter has to be set (unlike in the case of throwing an exception).

In this example, we just set it to the "default" int value, which is `0`. In the case of the generic version, we use the `default` operator to set the default value of a generic type `T`. Different concrete types can have different default values.

One thing to note here is that when we check the emptiness of the `list` argument, we do not check its "nullness". In fact, traditionally, we should have done something like this:

```
if (list is null || list.Count == 0) {
    // Throw an error or return.
}
// Use list here.
```

The boolean expression in the `if` statement does "short circuiting". If `list is null` (which is equivalent to `list == null`), then regardless of the value of the second operand of the OR (`||`) operator, the overall value is true, and hence the expression `list.Count == 0` is not evaluated, preventing the null reference exception.

In our sample, however, since we are using the enabled nullable context, the `list` argument of type `IList<int>` is guaranteed to be non-null. Hence, the null check is not needed. A nullable `int` list type would have been denoted as `IList<int>?`, with the question mark `?`.

The algorithm of finding the largest value in a list of `int` values is straightforward.

We start from the first element, and go through all the elements. Every time we see a value bigger than the maximum of the values that we have seen so far, we replace the current max with that value.

After having gone through the entire list, the current max is *the* max among all the elements in the list.

In the non-generic version, `TryFindMaxInt()`, we use the comparison `v > max`. The expression `v > max` is equivalent to `v.CompareTo(max) > 0` for types like the numeric types for which the greater-than (`>`) operator is defined.

The non-generic method `TryFindMaxInt()` is pretty much equivalent to what the compiler would have generated from `TryFindMax<T>()` when `T is int`.

Generics is often called the "templates" (in languages like C++) since it generates a concrete type using the generic definition as a "template".

One small difference that we note here is that in the generic version, the out parameter type `out T? max` is declared nullable, `T?`. This is because the `default` value can be null for reference types. Values of `int`, which is a value type, cannot be null.

> C# has a construct called the "nullable value types". In that context, `null` signifies the absence of a value, not the null reference. Although there are definitely situations where the nullable value types can be useful (and, this is again a concept borrowed from the functional programming, e.g., the algebraic data types), we do not recommend the use of the nullable value

> types. We do not use the nullable value types in this book.
>
> One other thing to note is that the relationship between a value type and its nullable value type is *rather different* from that between a reference type and its nullable counterpart. It is unfortunate that the terminologies used by the C# community are not very consistent, but learning these subtle, or not so subtle, differences is a part of the learning process.

Summary

We explored the generic collection types in .NET.

We also learned how to define our own generic method, with a type constraint. The `default` operator is used to get the "default value" of a given type.

> Historically, generics came to C# first, and the type constraint feature was later introduced into the existing generics framework. The name "type constraint" implies that it is something that is added to "constrain" or "limit" other things. This is not, however, how it really works in the modern C#. The type constraints are the integral part of generics. The old generics without the type constraints are merely the special cases.
>
> That is why we introduce the "full generics" first (with the type constraints) in this book, rather than introducing the special cases (without type constraints) and moving to the "full generics".
>
> Note that, in the examples of this lesson, we could not have defined the `TryFindMax<>()` function without using the type constraints. This is a general case, not the other way around.
>
> Other modern languages like Rust (with Traits) and Swift (with

> Protocols) follow this same paradigm. And, this is also how it works in the strongly-typed functional programming languages like Haskell.

11.3. Exercises

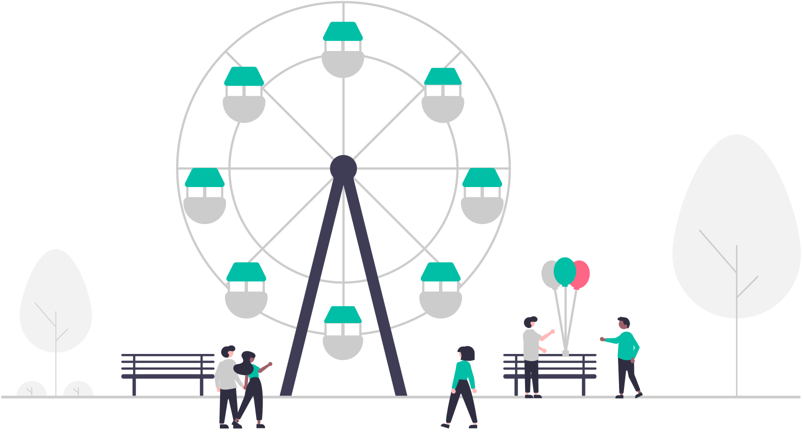
1. Write a generic function (static method) that finds the minimum value from a given list (without copying the example code).
2. Use a list of floating point numbers to test the function.
3. Write a generic function that finds the min and max values at the same time.
4. Write a generic function that finds the biggest even number from a list of integers that contain both even and odd numbers.

Chapter 12. Rotate Numbers

12.1. Agenda - Generic Methods, Generic Extension Methods, `IEnumerable`, `yield return`

We will review a few more C#'s basic concepts. In particular, we will cover `foreach` and `IEnumerable<T>`, and the `yield return` statement.

The sample code of this lesson also uses the generic methods which we learned in the previous lesson.



12.2. Code Reading - Array Rotation

Given an array, rotate or shift its array elements to the left by `k`. For instance, if we rotate an array, `{1, 2, 3, 4, 5}`, to the left by `1`, it will end up to be `{2, 3, 4, 5, 1}`. Note that the elements "wrap around", in this example.

The following main program demonstrates this behavior in a few different scenarios.

rotate-numbers/Program.cs

```
using Example;

var list = new List<int>() { 1, 2, 3, 4, 5 };
Console.WriteLine($"Input list:\t{list.Str()}");

var rotated2 = Rotator.Rotate(list, 7);
Console.WriteLine($"Rotated by 7:\t{rotated2.Str()}");

var rotated3 = Rotator.Rotate(list, -2);
Console.WriteLine($"Rotated by -2:\t{rotated3.Str()}");

var rotated4 = list.Shift(2);
Console.WriteLine($"Shifted by 3:\t{rotated4.Str()}");
```

The "Rotator.cs" file contains a couple of different implementations for rotating the elements in an array, one as a static method and the other as an extension method.

The rotator class also contains a couple of helper methods.

rotate-numbers/Rotator.cs

```
namespace Example;

static class Rotator {
    internal static IEnumerable<T> Rotate<T>(IList<T> list, int 
delta = 1) {
        for (var i = 0; i < list.Count; i++) {
            yield return list[Mod(i + delta, list.Count)];
        }
    }

    internal static IEnumerable<T> Shift<T>(this IList<T> list, int delta = 1) {
        for (var i = 0; i < list.Count; i++) {
            yield return list[Mod(i + delta, list.Count)];
        }
    }

    internal static string Str<T>(this IEnumerable<T> list) =>
        $"{{{string.Join(", ", list)}}}";

    private static int Mod(int n, int b) => ((n % b) + b) % b;
}
```

12.2.1. Explanation

The main program tests the implementations with a simple integer array, `{1, 2, 3, 4, 5}`.

```
$ dotnet run

Input list:     {1, 2, 3, 4, 5}
Rotated by 7:   {3, 4, 5, 1, 2}
Rotated by -2:  {4, 5, 1, 2, 3}
Shifted by 3:   {3, 4, 5, 1, 2}
```

12.2.2. Grammar

This is a small program, but there is a lot going on here.

All core implementations are included in the `Rotator` static class. Especially, the

`Rotator` class defines two static methods `Rotate()` and `Shift()`. (The method names are arbitrary. There is no difference between "rotation" and "shifting" in this example.)

Note the difference in which these two methods are called, in the main program. The `Rotate()` method is called just like any other static methods defined in a class. `Rotator.Rotate<int>()` takes a list of type `IList<int>` as its first argument.

On the other hand, the `Rotator.Shift<int>()` method is called as if it is a method defined on the type `IList<int>`, e.g., `list.Shift(2)`. There is no mention of `Rotator`, in this syntax, in which this method is defined.

This is called the "extension method", as we introduced in the previous lesson. The source code of the caller will still have to import the namespace of the class (using `using`) in which the extension method is defined, e.g., `Example` in this case.

The same with the method `Str()`. The `IList<int>` type does not have a method named `Str()` defined, and yet we use it, syntactically, as if it is a method of the type. The `Str()` extension method is defined as follows, in the `Rotator` class.

```
internal static string Str<T>(this IEnumerable<T> list) =>
    $"{{{string.Join(", ", list)}}}";
```

It is a generic method, with a type parameter `T`. There is no `where` type constraint. In fact, this method will work for all `IEnumerable<T>` types with any associated element type `T`.

Note that the implementation does not rely on any particular properties of type `T` although it uses the fact that the first argument is of type `IEnumerable<T>` (for any `T`).

The extension methods can be defined only in the static classes (i.e., the classes that cannot be instantiated).

The internal Str<T> static method takes one argument of type IEnumerable<T>, which is preceded by the keyword this. This indicates that this method is an extension method. That is, we can use this method as if it is one defined in the IEnumerable<T> type. The main program uses the syntax, list.Str(), to get the list's content as a string.

> Note that IList<T> *is* an IEnumerable<T>. That is, the IList<T> type is a subtype of IEnumerable<T> for the same element type T. (For example, IList<string> is a subtype of IEnumerable<string>.)
>
> We implement the Str() method on a slightly broader type IEnumerable<T>, rather than IList<T>, since that is all that is required by the implementation (e.g., string.Join()). In other words, the Str() implementation does not use any properties or methods specific to IList<T> other than the fact that it is a subtype of IEnumerable<T>.

The "extension method" is purely syntactic. One cannot add a method, or a property, to somebody else's type or static class. And, as stated, even though the name of the type or class is not explicitly used in the extension method syntax, its namespace has to be imported using the using declaration.

One of the most common namespaces that include a lot of extension methods is System.Linq from the .NET standard library, as we will see later in the book.

> In the earlier lessons, we stated that using the using declaration is optional. When using the extension methods, however, the using declaration is required. This is because there is no other way to explicitly specify the namespace in which the extension method is defined.

In C#, methods can be "overloaded". That is, different methods can be defined with the same name as long as these methods have different parameter list. This is true for both static and instance methods.

The `this` prefix in the parameter list, however, does not make the method signature different.

In this example, we have two methods, which perform exactly the same task, `IEnumerable<T> Rotate<T>(IList<T> list, int delta)` and `IEnumerable<T> Shift<T>(this IList<T> list, int delta)`. We could not have used the same name for these two methods because they essentially have the same method signature with the same parameter list.

Now, let's take a look at the implementation of the `Rotate<T>()` method:

```
IEnumerable<T> Rotate<T>(IList<T> list, int delta) { /* ... */ }
```

It is a generic method, without any type constraint (that is, no `where` clause). This method can be used with a list with any element type.

As stated before, an object of a type that implements `System.Collections.Generic.IEnumerable<T>` (or, `System.Collections.IEnumerable`) can be used in the `foreach ( in )` statement as a collection, among other things.

This `Rotate<T>` method returns an object of type `IEnumerable<T>`, and hence its return value can be used in the `foreach` iteration, and anywhere that requires an `IEnumerable<T>` object.

The interface `IEnumerable<T>` declares one method `IEnumerator<T> GetEnumerator()`. The `IEnumerator<T>` interface, in turn, declares a few properties/methods, including the `Current` property and the `MoveNext()` method.

In C#, the "yield return" syntax provides a shortcut for implementing

`IEnumerator<T>`.

Without going into too much detail, using the `yield return` statements hides the complexity of creating an object of the type `IEnumerator<T>`. Each `yield return` statement returns an element in an `IEnumerable<T>` collection object, so to speak.

> ℹ There are similar constructs in other languages as well. For example, it is called the "generators" or "generator functions" in Python and Javascript.

In the implementation of `Rotate<T>()`, we have a `for` loop in which we call `yield return` in each iteration. Since the loop iterates for `list.Count` times, the `IEnumerator<T>` return object from this method has `list.Count` elements, and its (implicit) `IEnumerator<T>` can iterate for the same `list.Count` times.

```
for (var i = 0; i < list.Count; i++) {
    yield return list[Mod(i + delta, list.Count)];
}
```

This is a special syntax in C#. The `yield return` (and, `yield break`) statements have broader uses. The methods using these `yield` statements can act like the "coroutines". This topic is, however, beyond the scope of this book.

The `Rotate<T>()` method returns, as its first element (`i == 0`), the value of `list[delta]` (ignoring the modulo operation for now). That is, the value of its first element (of the rotated list) is that of the `delta`-th element of the original list. The value of its second element, `i == 1`, is that of `list[1 + delta]` in the original list. And so forth. This effectively shifts the elements by `delta` to the left.

The result of `Mod(n, b)` function falls into a range from `0` (inclusive) to `b` (exclusive). (The double modulo operations in the implementation is to take care of the negative `n` values.)

The `IEnumerable<T> Shift<T>(this IList<T> list, int delta)` method is implemented exactly the same way other than that it is declared as an extension method.

Both methods use the default value of `1` for the second argument `delta`. That is, if the methods are used only with the first argument, then the default value `1` is used for the value of the second argument, `delta`, The method call `Rotate(list)`, for example, is equivalent to `Rotate(list, 1)`.

The arguments like `delta` that have the default values are often called the "optional arguments" since they do not always need to be specified. Only the last (consecutive) set of arguments in the argument list can be made optional.

Summary

We introduced the concept of method overloading. In C#, one can use the same method names for different (but related) methods as long as they have different parameter lists.

We looked at a couple of important interfaces in .NET, namely `IEnumerable<T>` and `IEnumerator<T>`, in this lesson. We also introduced a special syntax in C# to create a method that returns an `IEnumerable<T>`, using the `yield return` statement.

In addition, we reviewed the "extension methods", which we first introduced in the previous lesson.

12.3. Exercises

1. Create a (generic or non-generic) method that rotates an array of `strings` to the left by 2. For instance, given an input array `{"apple", "orange", "banana", "pair", "peach"}`, the method should return something like this:

`{"banana", "pair", "peach", "apple", "orange"}`.

2. Implement a method that rotates a list of `ints` to the left by `1`, "in-place", that is, without creating an additional list. The function signature might be something like this: `void ShiftByOne(int[] arr)`. Calling this function/method will shift the elements of the `arr` array to the left by `1`.

Chapter 13. Sort Numbers

13.1. Agenda - Generic Methods, Type Constraints, `IEquatable`, `IComparable`, `dynamic` Casting

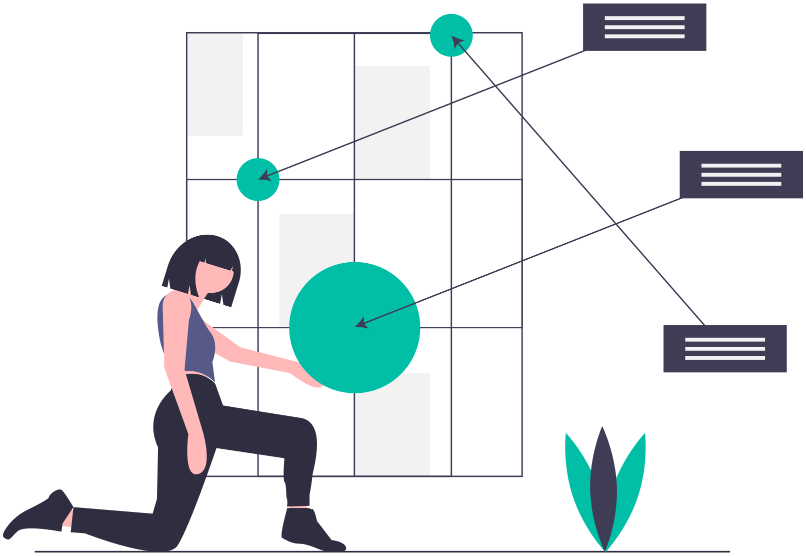
Sorting is one of the most common operations in programming. There are a lot of different algorithms that can sort, for example, a list of numbers in an ascending order (e.g., the smaller numbers first and the bigger numbers last).

Most programming language libraries/runtimes come with an extensive set of sorting-related library functions and methods. .NET is no exception. In many cases, you will not reinvent the wheel, and end up just using the .NET-provided (generic) types and methods for sorting and its related tasks.

In this lesson, we will try to "reinvent the wheel", just for fun. ☺ And, we will add a small twist to the good old sorting problem. More specifically, We will try to sort a list of integers based on its "distance" from a particular number, which we call the "pivot".

For example, given a list `{9, 3, 19, 1}` and a pivot `4`, the list can be sorted as `{3, 1, 9, 19}`, in the ascending order of the distance, since the distance between `3` and `4` is the smallest, and then between `1` and `4`, and so forth.

This is relatively simple to do for a list of `ints` or any other particular numeric types. In this lesson, however, we will try to implement the sorting method *generically*, in particular, across *all* numeric types, not just `ints`.



This is an optional lesson. If you are new to C#, or if you have no prior experience with generics in other programming languages, then it is best to skip this lesson in your first reading. As stated, for the common operations like sorting, you will more than likely just use the library functions. .NET also includes a number of interfaces like `IComparer<T>` and delegates like `Comparison<T>` to make sorting easier across the broad range of problems.

13.2. Code Reading - Generic Bubble Sort

This code sample is a bit "contrived", partly due to the limitations of the current C# generics. We will use the code merely as an example to illustrate some essential concepts of C#, including generics, not as a practical example.

C# 11.0, which is to be released later in 2022, will include a feature called the "static abstract method in interfaces". This solves the problem that we are addressing in this lesson.

The main program tests the static sorting method `Bubble.Sort()` defined in the `Example` namespace using a list of integers.

sort-numbers/Program.cs

```
using Example;

var arr = new int[] { 64, 34, 25, 12, 22, 11, 90 };

var pivot = 20;
Console.WriteLine($"Pivot = {pivot}");

Console.Write("Original array:\t");
Bubble.Print(arr);

Bubble.Sort(arr, pivot);

Console.Write("Sorted array:\t");
Bubble.Print(arr);
```

As stated, the `Bubble.Sort()` method is implemented generically.

sort-numbers/Bubble.cs

```
namespace Example;

static class Bubble {
    internal static void Sort<T>(IList<T> arr, T pivot) where T : IEquatable<T>, IComparable<T> {
        for (var i = 0; i < arr.Count - 1; i++) {
            for (var j = 0; j < arr.Count - i - 1; j++) {
                if (Compare(arr[j], arr[j + 1], pivot) > 0) {
                    (arr[j], arr[j + 1]) = (arr[j + 1], arr[j]);
                }
            }
        }
    }

    private static int Compare<T>(T l, T r, T pivot) where T : IEquatable<T>, IComparable<T> {
        var (dl, dr) = (Distance(l, pivot), Distance(r, pivot));
        return dl.Equals(dr) ? 0 : (dl.CompareTo(dr) > 0 ? 1 : -1);
    }

    private static T Distance<T>(T l, T r) where T : IComparable<T> =>
        (l.CompareTo(r) > 0) ? Subtract(l, r) : Subtract(r, l);

    private static T Subtract<T>(T l, T r) where T : notnull =>
        (l, r) switch {
            (sbyte il, sbyte ir) => (T)(dynamic)(il - ir),
            (short il, short ir) => (T)(dynamic)(il - ir),
            (int il, int ir) => (T)(dynamic)(il - ir),
            (long il, long ir) => (T)(dynamic)(il - ir),
            (nint il, nint ir) => (T)(dynamic)(il - ir),
            (float il, float ir) => (T)(dynamic)(il - ir),
            (double il, double ir) => (T)(dynamic)(il - ir),
            (decimal il, decimal ir) => (T)(dynamic)(il - ir),
            _ => (T)((dynamic)l - (dynamic)r),
        };

    internal static void Print<T>(IEnumerable<T> arr) =>
        Console.WriteLine($"{{{string.Join(", ", arr)}}}");
}
```

13.2.1. Explanation

Here’s a sample output:

```
$ dotnet run

Pivot = 20
Original array: {64, 34, 25, 12, 22, 11, 90}
Sorted array:   {22, 25, 12, 11, 34, 64, 90}
```

You can verify that they are indeed sorted based on the "distances" or "differences" of their values from the pivot point, `20`, in this example run.

13.2.2. Grammar

The C# programming language supports an extremely powerful generics. In fact, its big brother C++, which first introduced generics/templates to the main stream programming some 30 years ago, now follows C# in various respects when it comes to the features of generics.

> Incidentally, the C++20 standard includes a lot of powerful new features, including "concepts". If you haven’t used C++ in a while, or if you have never programmed in C++, then it is highly recommended to take a new/fresh look at the "modern C++".

Despite all the advanced features in generics, however, the current design of C# generics has some limitations.

For instance, there is currently no way to create a generic method or class that can be used as a template across all different numeric types. Or for the types that support certain primitive operations such as the addition or the multiplication.

Although it is not a big limitation by itself, because of the fact that numeric types and primitive operations are such a crucial part of programming, you will sometimes feel limited even with the C#'s advanced generics support.

For instance, there are almost a dozen numeric types in C#. You do not want to implement the same method for each numeric type (without using generics). That is the whole point of using the generic templates in the first place.

This is not limited to the number types. In general, there is no general way to specify a group of types that you have no control over, e.g., the builtin types, or the types from other third party libraries, etc. For the types that you have defined, you can always use the interface type constraints, among other things.

> Haskell, for instance, has a concept of "Typeclass" to group/specify a set of related types. The "more modern" imperative languages like Rust and Swift also include the similar concepts like `Trait` and `Protocol`. They are slightly different, and slightly more general, than C#'s interface-based type constraints. The interested readers are encouraged to explore more from other resources on this topic.

The sample code in this lesson demonstrates one way to create a generic method over all numeric types (and, possibly more).

This is a demonstration that it *can* be done, but not necessarily that it should be done this way. This sample code, along with all code samples in this book, is primarily used to illustrate the features of the C# programming language.

Let’s start from the main driver. This example main program creates an array of integers, and then it calls `Bubble.Sort()` to sort the numbers in the array according to their distances from a preset anchor point, `pivot == 20`.

As indicated, we implement our "sort according to the distance" method using generics. In particular, across all different numeric types. That is, even if we use an array of doubles, for instance, the method should work.

But, first let’s take a look at a non-generic version of the implementation, especially, for an integer array. The compiler will use the generic template that we present in this lesson to generate something similar when the method is specialized for `T` being `int`.

```
void Sort(int[] arr, int pivot) {
    for (var i = 0; i < arr.Length - 1; i++) {
        for (var j = 0; j < arr.Length - i - 1; j++) {
            if (Compare(arr[j], arr[j + 1], pivot) > 0) {
                (arr[j], arr[j + 1]) = (arr[j + 1], arr[j]);
            }
        }
    }
}
```

This is known as the "bubble sort algorithm". It is one of the simplest (but, not the most efficient) algorithms. It compares each pair of the adjacent elements, and whenever they are "out of order", it switches them so that they are "in order".

In this particular example, the order is defined by the distance of a given number to the pivot number.

Here’s a non-generic implementation of the `Compare()` static method, for integers:

```
int Compare(int l, int r, int pivot) {
    var (dl, dr) = (Math.Abs(l - pivot), Math.Abs(r - pivot));
    return (dl == dr) ? 0 : ((dl < dr) ? -1 : 1);
}
```

We use the `Math.Abs()` static method, from the `System` namespace, to compute the "distance" between two `int` numbers. Note the use of C# anonymous tuples in declaring and assigning two numbers at the same time.

The `Math.Abs()` function for an `int` argument behaves something like this, using the ternary operator:

```
int Abs(int num) => num > 0 ? num : -num;
```

Or, more generally, the "distance" between a pair of numbers can be defined as follows without using `Math.Abs()`:

```
int Distance(int l, int r) => (l - r) > 0 ? (l - r) : (r - l);
```

(Note the use of the subtraction (`-`) operation to compute the "distance".)

Then, the two distances between the two pairs of numbers, between `l` and `pivot` and between `r` and `pivot`, are compared using the two ternary operators. Note that the second operand of the first ternary operator is another ternary operator expression. None of the parentheses are required in this return statement.

It is conventional, for this type of "comparison" functions, to return `-1` if the first argument is smaller than the second, and return `1` if it's the reverse. If the two values are the same, then it returns `0`. This `Compare()` function follows that convention.

The int-version of `Compare()` is then used in our int-version of the `Sort()` method. The `if` statement would have been a simple integer comparison if we were to sort numbers in terms of their natural size. For example,

```
if (arr[j] > arr[j + 1]) { /* ... */ }
```

This condition, if evaluated to true, indicates that the value of the left element (from a pair of the two consecutive elements) is bigger than the value of the right element. That is, they are "out of order". In the sorted order (which is our goal), the right element of the pair should be bigger.

Therefore, if the boolean expression evaluates to `true`, then we swap the two numbers.

```
(arr[j], arr[j + 1]) = (arr[j + 1], arr[j]);
```

Again, notice the liberal use of the tuples across all code samples in this book. Without using the tuples, it could have been written as follows, using a temporary local variable.

```
var temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
```

The end result is the same.

In the problem of this lesson, we do not sort the numbers according to their natural order. Instead, we use a special comparison function, `Compare()`, to meet our special requirements.

```
if (Compare(arr[j], arr[j + 1], pivot) > 0) {
    (arr[j], arr[j + 1]) = (arr[j + 1], arr[j]);
}
```

But, the principle is the same. If the two consecutive elements are not in the desired order, according to our requirements, then we swap them. Otherwise, we leave them as is.

And, we do this for all pairs, and for all "sub-ranges", starting from the entire range.

```
for (var i = 0; i < arr.Length - 1; i++) {
    for (var j = 0; j < arr.Length - i - 1; j++) {
        // ...
    }
}
```

We will leave it as an exercise to the reader to see why these two nested loops are needed, and why we use these particular (different) ranges for the outer and inner loops, etc. The main focus of this book is teaching the C# language, not the algorithms.

But, one can easily convince oneself that after the first iteration of the outer loop (for `i == 0`), the last element is the biggest one (in terms of the distance from the pivot in our example) out of all the elements in the given array.

Through the inner iteration of swapping two consecutive elements (for `j`), we do not end up sorting all the elements. But, one thing is guaranteed. Each (potential) swap of a pair pushes the bigger element to the right. After going through the entire inner loop, from beginning to end, the last element is the biggest.

Now, in the next iteration of the outer loop (`i == 1`), we do not have to sort the entire array. We only have to sort the sub-array (the range of `j` when `i == 1`), leaving the last element alone. And, it continues until the sub-array consists of only one element, the first element.

After these nested iterations, we will end up with a completely sorted array.

In the "Big O notation", the bubble sort algorithm takes `O(N^2)` time for an `N` element array/list. That is, as the number of elements increases in an array, the sorting time increases quadratically. Not the most efficient sorting algorithm.

Just as a side note, this implementation can be slightly improved by keeping track of whether the remaining region is already sorted or not. Here's an example.

```
internal static void Sort(int[] arr, int pivot) {
    bool swapped;
    for (var i = 0; i < arr.Length - 1; i++) {
        swapped = false;
        for (var j = 0; j < arr.Length - i - 1; j++) {
            if (Compare(arr[j], arr[j + 1], pivot) > 0) {
                (arr[j], arr[j + 1], swapped) = (arr[j + 1], arr[j],
true);
            }
        }

        if (swapped == false) {
            break;
        }
    }
}
```

Again, we will leave it to the readers to understand how this implementation works.

Did the author already mention that he loves tuples? ☺

It is just a style or preference, but the author hopes that (at least some) readers appreciate the incredible efficiency of using tuples.

> Without the tuples, the `if` statement in this example would have required *four separate statements*, potentially decreasing the code readability (since the reader has to read more lines, and potentially has to scroll more across the computer monitor, etc.).

The `break` statement breaks out of the enclosing loop (in this case, the `for` loop with `i`), effectively ending the execution of the method without having to go through all the elements/subranges of the given array.

For completeness, here's an `int`-specific non-generic implementation of `Print<T>`.

```
void Print(int[] arr) => Console.WriteLine($"{{{string.Join(", ",
arr)}}}");
```

As explained before, the curly braces, which are used for variable substitution in the `$-interpolated string`, can be "escaped" by using two consecutive braces, e.g., `{{` and `}}`. Refer to the sample output in the beginning of the lesson to verify what they print out.

Since the implementation does not use any specific properties or methods of the type (`int` in this case), one can easily make this static method generic, without having to change the implementation:

```
void Print<T>(T[] arr) { /* ... */ }
```

The compiler will generate a method similar to the `int`-version of `Print()` based on this template when it sees that our program uses an `int` array.

If we use a decimal array (`decimal[]`) in the program, for instance, the compiler will also generate that version using the same template. We do not have to implement multiple versions of `Print()` for different array types. That is the power of generics.

The sample code uses a slightly more general argument `IEnumerable<T>` instead of `T[]` since that is all that is required by the implementation (e.g., `string.Join()`). Note that type `T[]` (e.g., a concrete type like `string[]`) implements an interface `IList<T>` (e.g., `IList<string>`), which extends `IEnumerable<T>` (e.g., `IEnumerable<string>`).

Now, let's go back to our generic implementation of `Sort<T>()`.

As seen in the `int` example, the types that we use will have to support the " equality" and "greater-than comparison" operations for us to be able to sort them according to our requirements.

That is,

```
void Sort<T>(IList<T> arr, T pivot) where T : IEquatable<T>,
IComparable<T> { /* ... */ }
```

As described before, the `where` clause is the type constraint for a generic type `T`. The constraint through the interfaces `IEquatable<T>` and `IComparable<T>`, from the `System` namespace, meets those requirements. As declared, the `Sort<T>` method can only used with the types that "implement" both `IEquatable<T>` and `IComparable<T>` interfaces.

Unfortunately, that is not enough, however. As we have seen in the `int` version of the `Sort()` method, the generic type will also have to support some kind of "subtraction" or "difference" operations. All numeric types support such operations (e.g., the binary minus operator). As stated in the beginning of the lesson, C# currently does not support such constraints for generic types.

There is really no good workaround. You can use a non-generic version (e.g., using `object` as a type), but that does not solve the problem either. (Numbers, or types that support "difference", are a small number of special types out of many possible types based on `object`.)

As promised, however, we will show you one example of how to do this without sacrificing generics too much.

The "bubble sort" implementation of generic `Sort<T>(IList<T> arr, T pivot)` is not much different from its non-generic `int` counterpart. The most important requirement is that the method uses a generic `Compare<T>()` method, which takes three arguments of type `T`, with the same "comparison" requirements.

The implementation of the `Compare<T>()` method relies on the fact that the type `T` supports `Equals<T>()` and `CompareTo<T>()` methods. This requirement is satisfied by the interface-based type constraint, `where T : IEquatable<T>, IComparable<T>`.

In addition, `Compare<T>()` uses the custom `Distance<T>()` method, which is essentially a generalization of `Math.Abs()`. The implementation of the `Distance<T>()` method uses another helper method, `Subtract<T>()`. This is where the "magic" happens. `Subtract<T>(T l, T r)` computes the different between two given argument values, `l` and `r`, *using the `-` operator*.

As indicated, not all types (even those that implement `IEquatable<T>` and `IComparable<T>`) support the binary subtraction operation. All numeric types support the minus operation. As we will see later in the book, one can define an implementation for the minus `-` operator for the custom types (known as the " operator overloading"), and those types can be used with our `Sort<T>()` method.

The implementation of `Subtract<T>()` seems quite complicated. Let's break it down.

First, the `notnull` constraint requires the type `T` should be non-nullable. This is fine in our case because our primary uses will be for numeric types and, possibly, for other custom values types (which cannot be null).

Then, the method uses the "expression body". That is, it is one large expression. As stated, whether ones uses an expression body or a block body is a matter of taste and preference in most cases.

C# has an interesting feature called the "switch expression". This is not to be confused with the traditional "switch statement", which is found in virtually all C-style languages. The `switch` expression feature of C# is "borrowed" from the functional style programming languages. It uses "pattern matching", which is also a concept from the functional programming.

A general syntax is like this:

```
<expression> switch {
    <pattern1> => <value1>,
    <pattern2> => <value2>,
    // ....
    _ => <defaultvalue>,
}
```

In this notation, `switch` is the only keyword. In addition, it uses the "fat arrow" (`⇒`), comma (`,`), underscore (`_`), and a pair of curly braces (`{` and `}`).

Despite the use of the curly braces, it is an expression, not a statement. It cannot be used as a stand-alone statement. In the sample code, the value of the switch expression is used in the expression body. If we had used a block body, then we would have used `return` to return this value.

The `switch` expression works as follows: The type, value, and other properties, etc. of `<expression>` are computed first. Then they are "matched" against a series of patterns in the switch expression. If it matches any of the patterns, by going through the patterns from top to bottom, then the value of the entire expression is the value specified on the right hand side of `⇒` for that matched pattern.

The patterns in the switch expression have to be "exhaustive". That is, the list of the patterns has to cover all possible cases of the `<expression>`.

The optional _ identifier like a wildcard. It can be used as a "catch-all" case. If none of the explicit patterns matches, then the value of the default case is used.

C#, as of 9.0, supports many kinds of "patterns". It even supports Boolean combinations of all different patterns, using `and`, `or`, and `not`,

In this sample code, we use the type patterns. (Refer to the C# language reference for all the possible pattern types.) The `<expression>` is an anonymous tuple, `(l, r)`, in which `l` and `r` are the arguments of type `T` passed in to the `Subtract<T>()` method.

The first pattern, `(sbyte il, sbyte ir)`, for instance, will match if the type `T` is `sbyte`. When it matches, the variables `il` and `ir` of type `sbyte` can be used for `l` and `r`, respectively. This syntax makes it unnecessary to explicitly have to cast these variables.

If the third pattern matches because `T` is type `int`, for instance, then the overall value of the `switch` expression is `(T)(dynamic)(il - ir)`.

Notice the double casting. The expression `il - ir` (which is the value of `il` subtracted by `ir`) is a value of `int` type, in this specific case. But, we still need to cast it to `T` to satisfy the method signature.

The compiler at this point does not know anything about the relationship between the types `T` and `int`. Hence, we will need to cast it to the `dynamic` type first, and then it can be cast to the desired type `T`. At runtime, if the third pattern is matched, then `T` is `int`. But, the compiler does not know that at build time.

The keyword `dynamic` allows us to bypass the compiler’s static type checking, and its use is generally not recommended. We are doing this in this sample code only because of the limitations of the generics in C#.

We do this for all signed integer and floating point types. When a pattern matches, we use the native minus operations for that specific type.

For any other types (for the `_` default pattern), we simply compute its "minus" value, `(T)( (dynamic)l - (dynamic)r )`, and hope for the best. If the type `T` happens to have a minus (`-`) operation defined, then it will work. Otherwise it will throw a runtime exception.

We could have just thrown an exception, but doing so would have prevented us from using this method for any (custom) types other than the (already-known) numeric types.

In this sample code, we just take a risk and try to compute `(T)((dynamic)l - (dynamic)r)`. Clearly, the caller of this method should know what to expect when they use non-numeric types. In this expression, notice again the double casting. Without `dynamic`, the compiler would not let us compute the subtraction of the two values of an arbitrary type `T`.

> In this particular implementation, we are supporting only the signed integer types through the explicit patterns.
>
> The example is just for illustration, and in fact, we could do away with the whole switch expression since at the end of the day, we are computing `(T)((dynamic)l - (dynamic)r)` for types. E.g., `T Subtract<T>(T l, T r) => (T)((dynamic)l - (dynamic)r);`. (If we throw an exception for non-matched types, or do something else, then these explicit patterns are required.)
>
> If we want to include all unsigned integer types as patterns as well, what changes would be necessary in the implementation of `Sort<T>()` and its helper methods?
>
> (Note that the reason why we could not list unsigned integer type patterns in this particular implementation was because the minus

> operation could result in a value of signed integer types.)

Now that we have the implementation of `Compare<T>()` with the help of `Distance<T>()` and `Subtract<T>()`, the implementation of `Sort<T>()` is complete.

As shown earlier, we can easily verify this implementation using an array or list of any type `T` which is an `IEquatable<T>` and an `IComparable<T>` and which supports the minus (`-`) operation. The demo program merely uses an array of `int` type for simplicity.

Summary

We explored generics a little bit deeper in this (optional) lesson. As stated, the current "limitation" of the C# generics, as illustrated in this chapter, will be removed in C# 11.0, when the "static abstract method in interfaces" feature is officially introduced. It is very likely that the next version of .NET will also include a number of helper interfaces to make it easier to represent certain type constraints, such as `INumber` for instance.

> You can try this feature in .NET 6/C# 10.0 by enabling the "preview" option. Refer to other resources if you want to try it out before the C# 11.0 release.

In this lesson, we also introduced the `switch` expression of C# 8.0. Although it can be viewed as a simple generalization of the ternary expression and the switch statement, its pattern matching capability makes it more flexible and more expressive. We will likely see more and more uses of the `switch` expressions in Modern C#.

13.3. Exercises

1. Although the bubble sort algorithm is one of the easiest sorting algorithms, it is rarely used in practice because it is not very efficient. Another sorting algorithm that is easy to implement and that is widely used is the so-called quick sort algorithm. Try implementing the `void Sort<T>(IList<T> arr, T pivot) where T : IEquatable<T>, IComparable<T>` method using quick sort.

Chapter 14. Morse Code

14.1. Agenda - `Dictionary`, `Dictionary` Initializers, Static Constructors, `StringBuilder`

Morse code is a method for encoding a set of alphabets and numbers used in telecommunications. It uses a combination of two types of signals, short (or, "dot") or long (or, "dash"), along with various length "gaps", to represent characters, and words.

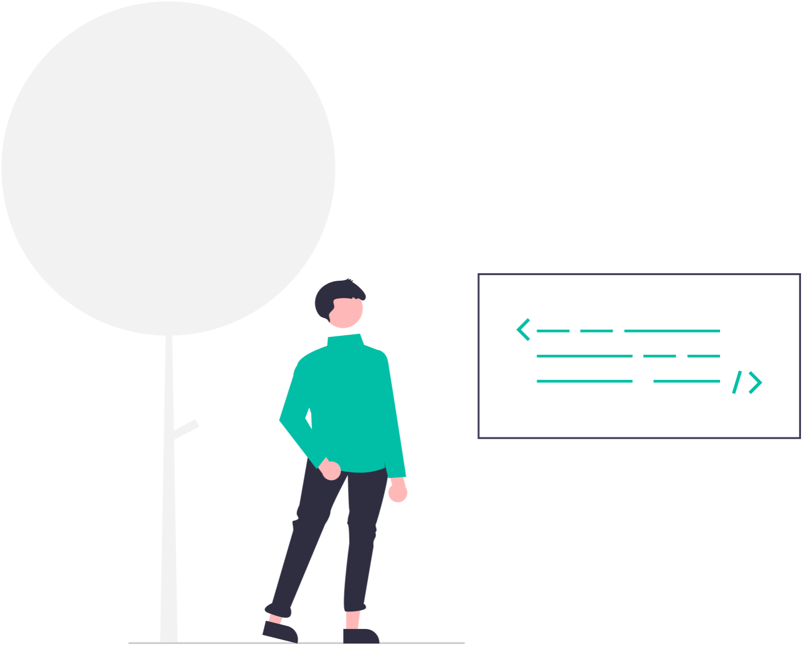
For more information, refer to other resources on the Web. For example, the wikipedia page: en.wikipedia.org/wiki/Morse_code

In this lesson, we will write an "encoder" (alphabets to Morse code) and a "decoder" (Morse code to alphabets) in the C# programming language. This is not a realistic program, and it is not intended to be a practical example. However, the concepts presented here will be useful in general programming, especially in the context of communications.

The program is primarily written as a library, and the main functionality is included in the `Morse` namespace. The "main program" in this example, as in many projects in this book, is mainly used as a "quick and dirty" test driver.

We can easily change the project type to the `library` assembly if hypothetically we want to share this with other developers (in the world).

> ℹ We will discuss the "NuGet" packages later in the book.



14.2. Code Reading - Morse Code Encoding and Decoding

We have two public static classes, `Encoder` and `Decoder`, which have public static methods `Encode()` and `Decode()`, respectively.

The main program calls these two methods with some sample data, and just prints out the results for visual inspection.

morse-code/Program.cs

```
using Morse;

var text1 = "Hello, World!";
var code1 = Encoder.Encode(text1);
```

```
Console.WriteLine($"text: {text1} => code: {code1}");

var code2 = ".... . .-.. .-.. --- --..--     .-- --- .-. .-.. -.. -.-.--";
var text2 = Decoder.Decode(code2);
Console.WriteLine($"code: {code2} => text: {text2}");
```

For this example, we simply use strings for both English text and Morse code. We define a mapping from alphabets (and numbers and punctuations) to Morse code, and also its reverse mapping.

morse-code/Code.cs

```
namespace Morse;

public static class Code {
    public static Dictionary<char, string> MorseCode {
        get => morseCode;
        set => morseCode = value;
    }
    public static Dictionary<string, char> ReverseCode {
        get => reverseCode;
        set => reverseCode = value;
    }

    private static Dictionary<char, string> morseCode = new() { };
    private static Dictionary<string, char> reverseCode = new() { };

    static Code() {
        foreach (var (k, v) in Chars.Code) {
            ReverseCode[v] = k;
            MorseCode[k] = v;
            if (char.IsLetter(k)) {
                MorseCode[key: char.ToUpper(k)] = v;
            }
        }
    }
}
```

`static Code() {}` is a "static constructor", used to initialize two static variables, `morseCode` and `reverseCode`. Note that these are variable of the "Dictionary type".

Here’s a mapping from alphabets to Morse code symbols:

morse-code/Chars.cs

```
namespace Morse;

public static class Chars {
    public static Dictionary<char, string> Code {
        get => code;
        set => code = value;
    }

    private static Dictionary<char, string> code = new() {
        ['a'] = ".-",
        ['b'] = "-...",
        ['c'] = "-.-.",
        ['d'] = "-..",
        ['e'] = ".",
        ['f'] = "..-.",
        ['g'] = "--.",
        ['h'] = "....",
        ['i'] = "..",
        ['j'] = ".---",
        ['k'] = "-.-",
        ['l'] = ".-..",
        ['m'] = "--",
        ['n'] = "-.",
        ['o'] = "---",
        ['p'] = ".--.",
        ['q'] = "--.-",
        ['r'] = ".-.",
        ['s'] = "...",
        ['t'] = "-",
        ['u'] = "..-",
        ['v'] = "...-",
        ['w'] = ".--",
        ['x'] = "-..-",
        ['y'] = "-.--",
        ['z'] = "--..",

        ['0'] = "-----",
        ['1'] = ".----",
        ['2'] = "..---",
        ['3'] = "...--",
        ['4'] = "....-",
        ['5'] = ".....",
        ['6'] = "-....",
        ['7'] = "--...",
        ['8'] = "---..",
        ['9'] = "----.",

        ['.'] = ".-.-.-",
        [','] = "--..--",
        ['?'] = "..--..",
        ['\''] = ".----.",
        ['!'] = "-.-.--",
        ['/'] = "-..-.",
        ['('] = "-.--.",
        [')'] = "-.--.-",
        ['&'] = ".-...",
            [':'] = "---...",
            [';'] = "-.-.-.",
            ['='] = "-...-",
            ['+'] = ".-.-.",
            ['-'] = "-....-",
            ['_'] = "..--.-",
            ['"'] = ".-..-.",
            ['$'] = "...-..-",
            ['@'] = ".--.-.",
        };
}
```

The Chars static class has a read-write "static property", Code, which is of the type Dictionary<char, string>.

"Encoding" a text amounts to mapping the alphabets of the text to the corresponding strings representing the Morse code. In practice, Morse code involves a few different length of gaps, etc. We will represent those gaps with spaces.

The implementation of Encoder.Encode() is straightforward.

morse-code/Encoder.cs

```
namespace Morse;
using System.Text;

public static class Encoder {
    public static string Encode(string text) {
        var sb = new StringBuilder();

        foreach (var c in text) {
            if (Code.MorseCode.ContainsKey(c)) {
                var v = Code.MorseCode[c];
                sb.Append(v).Append(' ');
            } else {
                if (c == ' ') {
                    sb.Append("   ");
                } else {
                    sb.Append("???");
                }
            }
        }

        return sb.ToString();
    }
}
```

If a character is not representable by a Morse code, then we simply output "???" in this example.

Implementing the `Decode()` function requires a little more thinking. This is not necessarily an artifact of our toy example. Decoding Morse code is inherently more complicated than encoding. This is because while the unit of a signal is dots and dashes (and gaps) it is a series of these signals (one or more) that represent a single character in English.

The following `Decoder.Decode()` function implements one of the simplest solutions. One can probably implement this more efficiently using a better algorithm.

morse-code/Decoder.cs

```
namespace Morse;
using System.Text;

public static class Decoder {
    public static string Decode(string cipher) {
        var sb = new StringBuilder();

        var code = new StringBuilder();
        var spaceCount = 0;
        foreach (var c in cipher) {
            if (c == ' ') {
                ++spaceCount;
            } else {
                if (spaceCount > 0) {
                    AddChar(sb, code);

                    if (spaceCount > 1) {
                        sb.Append(' ');
                    }
                    spaceCount = 0;
                }
                code.Append(c);
            }
        }
        AddChar(sb, code);

        return sb.ToString();
    }

    private static void AddChar(StringBuilder sb, StringBuilder code) {
        if (code.Length > 0) {
            try {
                var ch = FindChar(code.ToString());
                sb.Append(ch);
            } catch (Exception ex) {
                Console.WriteLine(ex);
                sb.Append('?');
            }
        }
        code.Clear();
    }

    private static char FindChar(string code) {
        if (Code.ReverseCode.ContainsKey(code)) {
            return Code.ReverseCode[code];
        } else {
            throw new ArgumentException($"Given code {code} not found.");
        }
    }
}
```

This particular implementation relies on the assumption (specific to this example) that one space is used between characters and more than one spaces are used between words.

14.2.1. Explanation

If you run the program as before:

```
dotnet run
```

You get the following output:

```
text: Hello, World! => code: .... . .-.. .-.. --- --..--    .-- ---
.-. .-.. -.. -.-.--
code: .... . .-.. .-.. --- --..--    .-- --- .-. .-.. -.. -.-.-- =>
text: hello, world!
```

14.2.2. Grammar

Let’s start from the beginning.

When you are given a problem, as a general rule, it is best to solve the problem from top to bottom. That is, from the high-level organization, description, understanding, etc. to the lower-level details. The software "design" is often done this way. Then, you start to implement the lower level components first and build upward.

This is a general guideline in solving computational problems. This kind of strategy may not work in all problem domains.

In this particular example, we may need to read an input in some form (presumably, converted from the Morse code signals, electrical or otherwise), and convert them into English. On the flip side, we can convert an English text to a sequence of symbols in certain formats (which can be read and converted to Morse code signals in some way, for instance).

Without considering specific (executable) programs, therefore, we can imagine that we will need the following "API" to support a broad range of programs.

```
string Encode(string text);
string Decode(string cipher);
```

In this particular example, as with most examples in this book, we do not have well-defined, and specific, requirements. In practice, we will most likely start from a set of particular requirements, which will likely constrain our "API design". In this lesson, we will use the use cases of the `Main()` method as a requirement. Then, this API design will most likely support those use cases.

Now, how do we implement the `Encode()` method?

As stated, "encoding" is simply a "dictionary lookup", in this example. The C# programming language, and .NET, provides a generic data type `Dictionary<K,V>`, which we can use for this purpose.

A dictionary stores key-value pairs, and it provides a way to retrieve the value corresponding to a given key. As stated, all collection types are reference types.

The Dictionary field, `Chars.code`, is initialized with all characters relevant to English. Note the collection initialization syntax for a `Dictionary` object, which uses a syntax similar to an assignment.

The Morse Code scheme does not distinguish the upper and lowercase letters. For convenience, we create a new `Dictionary` that includes both lowercase and uppercase Alphabets in the static constructor of the `Code` class.

This new Dictionary initialized in the static constructor of `Code` is named `MorseCode` in this example.

The `Encoder.Encode()` method uses an object of `StringBuilder`. It goes through each character in the given `text`, and if it has a corresponding Morse code (`Code.MorseCode.ContainsKey(c) == true`), then the code is added to the `StringBuilder` object. If not, it adds three spaces for a space (`' '`) or an invalid value, `"???"`.

The accumulated string is then returned via the `ToString()` method.

Writing, or understanding, the `Decoder.Decode()` function requires a little bit more thinking. We will leave this as an exercise to the reader.

But, it essentially follows the same logic as `Encode()`. It starts with a Dictionary `ReverseCode` and it iterates over the character array of the input `cipher`. The only difference is that we will need to read possibly more than one chars (until the next space) to match the code to a corresponding letter.

Summary

We reviewed the generic `Dictionary<K,V>` collection type in this lesson. A `Dictionary` is a reference type.

A dictionary stores key-value pairs. C# uses mostly similar syntax to those used in other programming languages for similar data types.

14.3. Exercises

1. Study the implementation of the `Decoder.Decode()` static method in this lesson, and implement the same functionality from memory (without directly copying the method).
2. Write a program that reads an input in Morse code and prints out a warning every time it sees a code corresponding to "SOS" in the input.

Chapter 15. Mega Millions

15.1. Agenda - Value Tuples, Enums, Random Number Generator, `break` and `continue` Statements

In many countries, lotteries are the monopolies of the government. The lotteries are typically used to raise money to fund the public projects.

In the United States, there are a few "mega" lotteries whose tickets are sold across multiple states, and whose jackpots often reach over hundreds of millions of dollars. The biggest ones are Powerball and Mega Millions.

Let’s play a game of the "fake" Mega Millions using the C# random number generator.

The sample code of this lesson is based on the information from the official Mega Millions website, www.megamillions.com/, in particular, from this page, How to Play [https://www.megamillions.com/How-to-Play.aspx].

15.2. Code Reading - Lottery Simulation

The main program is rather simple. All the game logic is implemented in the static method `Millions.Play()` in the namespace `Mega`.

mega-millions/Program.cs

```
Console.WriteLine($"Let's play Fake Mega Millions (for fun)!");
```

```
Mega.Millions.Play();
```

Here’s the implementation of the `Millions` static class:

mega-millions/Millions.cs

```
namespace Mega;

public static class Millions {
    public static void Play() {
        var (numbers, mega) = Input.Read();
        Console.WriteLine($"You selected: numbers = [{string.Join(',', numbers)}]; Mega number = {mega}");

        var (whites, gold) = Game.DrawBalls();
        Console.WriteLine($"White balls drawn = [{string.Join(',', whites)}]; The golden ball = {gold}");

        var prize = Game.DeterminePrize(whites, gold, numbers, mega);
        if (prize > Prize.None) {
            Console.WriteLine($"Congratulations! You won!!! Your prize is {prize}!");
        } else {
            Console.WriteLine($"Thanks for playing Fake Mega Millions!");
        }
    }
}
```

Note that we declare the Millions class `public` in this lesson. Although it is an `Exe` assembly that we are building here, we can easily change it to the library type if we want to make this game library available to other C# developers.

This class includes one static method, `Play()`, which is also declared as `public`.

The static class, `Input`, handles the user input:

mega-millions/Input.cs

```
namespace Mega;

public static class Input {
    public static (ushort[], ushort) Read() => (
         ReadFiveNumbers(),
         ReadMegaNumber()
    );

    private static ushort[] ReadFiveNumbers() {
        Console.Write("Select 5 numbers between 1 and 70: ");

        var numbers = new ushort[Game.NumWhiteBalls];
        var i = 0;
        while (i < Game.NumWhiteBalls) {
            try {
                var parts = Console.ReadLine()!.Split();
                foreach (var part in parts) {
                    if (ushort.TryParse(part, out ushort number) && IsValidNumber(number)) {
                        numbers[i++] = number;
                    }
                    if (i >= Game.NumWhiteBalls) {
                        break;
                    }
                }
            } catch (Exception) {
                continue;
            }
        }

        Array.Sort(numbers);
        return numbers;
    }

    private static ushort ReadMegaNumber() {
        Console.Write("Select 1 mega number between 1 and 25: ");

        ushort mega;
        while (true) {
            var input = Console.ReadLine();
            if (ushort.TryParse(input, out ushort number) && IsValidMegaNumber(number)) {
                mega = number;
                break;
            } else {
                Console.WriteLine("Input a number in the range from 1 to 25.");
                continue;
            }
        }

        return mega;
    }

    private static bool IsValidNumber(ushort num) => (num >= 1) && (num <= 70);
    private static bool IsValidMegaNumber(ushort num) => (num >= 1) && (num <= 25);
}
```

The lottery game logic is included in the `Game` static class:

mega-millions/Game.cs

```
namespace Mega;

public static class Game {
    internal const int NumWhiteBalls = 5;

    private static readonly Random rand;

    static Game() => rand = new Random();

    public static (ushort[], ushort) DrawBalls() {
        var whites = new ushort[NumWhiteBalls];
        for (var i = 0; i < NumWhiteBalls; i++) {
            whites[i] = (ushort)(rand.Next(75) + 1);
        }
        var gold = (ushort)(rand.Next(25) + 1);

        Array.Sort(whites);
        return (whites, gold);
    }

    public static Prize DeterminePrize(
        ushort[] whites,
        ushort gold,
        ushort[] numbers,
        ushort mega
    ) => (whites.Count(numbers.Contains), gold == mega) switch {
        (0, true) => Prize.Mega,
        (1, true) => Prize.OnePlusMega,
        (2, true) => Prize.TwoPlusMega,
        (3, false) => Prize.Three,
        (3, true) => Prize.ThreePlusMega,
        (4, false) => Prize.Four,
        (4, true) => Prize.FourPlusMega,
        (5, false) => Prize.Five,
        (5, true) => Prize.Jackpot,
        _ => Prize.None,
    };
}
```

Finally, the file, *Prize.cs*, includes the definition of an "enum type", `Prize`:

mega-millions/Prize.cs

```
namespace Mega;

public enum Prize : byte {
    None = 0,
    Mega,
    OnePlusMega,
    TwoPlusMega,
    Three,
    ThreePlusMega,
    Four,
    FourPlusMega,
    Five,
    Jackpot, // Five + Mega
}
```

Try to read and understand the program even if you do not know the rules of the game, Mega Millions.

"Reading" is generally more difficult than "writing". If you are new to programming, then the (long) code samples like this can be intimidating. *Where do I start?*

As you might have noticed, we always try to list the code samples

> "from top to bottom" in each lesson. It is generally the best way to "read" the code, that is, to start from the top and go down, or go "deeper". For example, the game's high level logic is all included in the `Millions` class in this example. Even if you do not know how exactly the methods like `Game.DrawBalls()` and `Game.DeterminePrize()` are implemented, you can get the general sense of how the overall program works just by looking at the `Millions.Play()` method (which is called from our "main program").
>
> In fact, the `Play()` static method includes four "parts":
>
> - Reading the user input, e.g., `Input.Read()` (line 5).
> - Drawing the balls, e.g., `Game.DrawBalls()` (line 8).
> - Checking the user input against the drawn balls, e.g., `Game.DeterminePrize()` (line 11).
> - And, finally, outputting the game result, e.g., lines 12 - 16.
>
> Now, you can examine each of these parts in turn, possibly going through different classes, and different source files, etc. And, you go further and further down, if you need to, when you read large programs.

15.2.1. Explanation

As before, we just use `dotnet run` during the development, rather than using the commands `dotnet build` or `dotnet publish`.

The program starts by asking the player for 6 numbers, 5 for the white balls and 1 for the Mega ball.

Here's a sample run:

```
$ dotnet run

Let's play Fake Mega Millions (for fun)!
Select 5 numbers between 1 and 70: 3 19 65 23 7            ①
Select 1 mega number between 1 and 25: 10                  ②
Yor selected: numbers = [3,7,19,23,65]; Mega number = 10
White balls drawn = [1,21,22,45,73]; The golden ball = 17
Thanks for playing Fake Mega Millions!
```

① A user input of 5 numbers from 1 to 70.

② Another input of a number from 1 to 25.

The program prints out the text "Thanks for playing Fake Mega Millions!", gently letting the player know that he/she did not win any prize ☺, and it terminates.

Here's another sample run, in which the player has won a prize:

```
$ dotnet run

Let's play Fake Mega Millions (for fun)!
Select 5 numbers between 1 and 70: 22 33 11 66 55
Select 1 mega number between 1 and 25: 15
You selected: numbers = [11,22,33,55,66]; Mega number = 15
White balls drawn = [4,9,22,46,59]; The golden ball = 15
Congratulations! You won!!! Your prize is OnePlusMega!
```

The probability of winning *any prize* is pretty small. You can easily (at least approximately) compute the odds of winning a prize, and it's less than 5%. The author had to play this game many times to get the one win (so that he could include its output in the book). It was the most difficult part of writing this book. 😀

15.2.2. Grammar

An `enum` type is a value type. It defines a custom integral numeric type. The enum types are used essentially to define a set of (related) named integer constants.

To define an enum type, we use the `enum` keyword and specify the names of enum members:

```
enum Prize {
    Mega,
    // ...
}
```

By default, the associated constant values of the enum members are of type `int`. You can explicitly specify any other integral numeric type as the underlying type of an enum type. For example,

```
enum Prize : byte {
    Mega,
    // ...
}
```

The enum values start with zero by default and increase by one in the order defined. One can also explicitly set a specific constant value for each enum member, if desired.

```
enum Prize : byte {
    Mini = 10,
    Mega = 100,
    Super,
    // ...
}
```

In this example, the value of `Super` is `101` following the "increase by one" rule (because it is assigned no specific value and the value of its preceding member is `100`).

The default value of an enum type `E` is the value corresponding to an int value `0` (even if zero doesn’t have the corresponding enum member). It is a good practice to define a default value for an enum type (or, for any value type). In the sample code, we define `Prize.None` as the default value (`0`) .

We cannot define a method inside the definition of an enum type. To add functionality to an enum type, we use the extension methods.

The `System.Enum` class of .NET, from which all custom `enum` types (implicitly) inherit, defines a number of useful methods, including the `Parse()` and `TryParse()` methods that we have seen before (in numeric types). Other commonly used methods are, `IsDefined()`, `GetName()`, `GetNames()`, and `GetValues()`. Refer to the .NET online documentation for more information, if you are interested.

A switch expression is used to evaluate a single expression from a list of candidate expressions based on the pattern match with an input expression.

For example, the `Game.DeterminePrize()` method uses a switch expression, based on an input expression, `(whites.Count(numbers.Contains), gold == mega)`, which is a tuple literal of type `(int, bool)`. This switch expression has nine "arms" for possible match expressions, one for each `Prize`, separated by commas (`,`), and one "catch-all" arm (`_`) in case none of the patterns matches.

The wildcard catch-all pattern should be the last one. Why?

The result of a switch expression is the value of the expression of the first switch expression arm whose pattern matches the input expression and whose case guard, if present, evaluates to true. The switch expression arms are evaluated in the order specified, from beginning to end.

Even though you may not know the exact syntax of the switch expression, you can still intuitively see what is happening in this switch expression.

A "case guard" is an additional condition that must be satisfied together with a matched pattern. You specify a case guard after the `when` keyword that follows a pattern, Although we do not use the case guards in any of the example code in this book, they can be very useful in differentiating similar patterns based on different conditions. Refer to the C# language reference for more information.

Note that the entire definition of the method, `Game.DeterminePrize()`, is included in a single expression. (As mentioned, the `return` statement is implied in the expression body.) If you are coming from the more traditional (C-style) programming style, then it may require some getting-used-to.

It is a matter of preference, but as stated, this is the future of C#. The modern C# prefers using expressions over statements, and the more compact over the more verbose.

As for the strange-looking method call `whites.Count(numbers.Contains)`, `T[].Count()` is a LINQ method.

It is an extension method defined in the System.Linq namespace, and its function signature is something like this: `int Count<T> (this IEnumerable<T> source, Func<T,bool> predicate)`. Note that we can use this method with `whites` because the type `ushort[]` is a subtype of `IEnumerable<ushort>`.

In this example, the `whites.Count()` method is called with the argument, `numbers.Contains` (without the parentheses), as a delegate of type `Func<T,bool>`. The `System` namespace defines a set of delegates, and the delegate `Func<T,bool>` represents a method that takes one parameter of a generic type `T` and returns a

Boolean value. In all `Func<…>` style delegates, the last type is the return type.

The `List<T>.Contains(T item)` method returns `true` if the `item` is in the list or `false` otherwise. Again, `ushort[]` is a subtype of `List<ushort>`. The `Count()` method goes through all elements in the `whites` array, and it counts those that are contained in the `numbers` array. Hence, the method call `whites.Count(numbers.Contains)` ends up returning the number of balls that are both in the `whites` and `numbers` arrays.

> The method like `Count<T>(…)` is often known as the "higher order function" because it is a function (method in C#) that take a function argument (delegate in C#).
>
> The System.Linq namespace includes a set of commonly used methods that are higher order functions. We will not go through all these methods in this book, but the readers are encouraged to consult other references. For example, refer to the Microsoft doc for a list of (extension) methods defined on the `IEnumerable<T>` Class [https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable?view=net-5.0].

The `System.Random` class represents a pseudo-random number generator, which is an algorithm that produces a sequence of numbers that meet certain statistical requirements for randomness.

The `Random` class defines a number of different methods to get random int, byte, or real numbers. The static `Game.DrawBalls()` method uses the `Random.Next(int)` instance method to get a non-negative random integer that is less than the specified maximum. That is, in this particular case, `rand.Next(25)`, for example, will return a random integer between `0` (inclusive) and `25` (exclusive). (25 possible values in total.)

Note that we add the number `1` to make the numbers fall within the range of 1 and

25 (both inclusive). We could have achieved the same thing using a different method `Random.Next(int, int)`. How exactly? We will leave it to the readers since it is a good exercise to look up the API docs.

An object of `Random` is instantiated in the static constructor of `Game`, in this example.

```
static Game() => rand = new Random();
```

We could have initialized the `rand` static field at the point of declaration:

```
private static readonly Random rand = new Random();
```

In this case, there is really no difference. Typically, the static constructors are used to do certain "more complicated" initializations that cannot be accomplished using the simple static initializer syntax.

Note that the `DrawBalls()` method returns a tuple type to return "multiple values", so to speak. It is generally preferred to use the tuples rather than to use one or more `out` parameters for this purpose.

We use `ushort` to represent the ball numbers from 1 to 25 and from 1 to 70 in this example (for illustration). We could also have used `byte` since its range covers all the numbers that we are using.

In general, however, it is not worth the time and effort to "optimize" this type of things. It is often good enough just to use an `int` for integer types unless there is a special performance or storage constraint. In some cases, the possible integer range that we are dealing with may be beyond that of the 32 bit `int` type. And, we will have to use `long` or `ulong`.

One thing to note is that this practice of the "32-bit int as the default integer type" started years ago. (C# was created some 20 years ago.)

These days, most machine architectures use 64 bits. Some programming languages support 128 bit integer types. It is probably a time that we used the 64 bit int as the default integer type.

> C# now has `nint` and `nuint` types, as indicated earlier, whose widths are 64 bits on the 64 bit machines. It is not entirely clear, to the author, however, how the C# community will adopt these new types. As currently introduced, these types are mainly to be used in the "interop" scenarios, or in the "low level" programming.

One last thing to note in the `DrawBalls()` method is the use of the sorting method `Array.Sort()`. All collection types in .NET support a number of (overloaded) sorting methods. The one that we use from the `Array` class is a generic method that is specialized to `ushort`, `Array.Sort<ushort>(whites)`. As stated, the type parameter can be omitted if the type can be inferred.

As for the the implementation of the `Input` static class, we will leave it to the readers as a reading exercise. For example, why do we use the nested loops, `foreach` within `while`, in the `ReadFiveNumbers()` method? What does it accomplish?

Just remember that the code samples included in this book are not necessarily "the best" code (even if such a thing exists). One can implement the same functionalities in many different ways.

It is always a good exercise to think about possible alternative designs and implementations when you read somebody else’s code.

Summary

We used an instance of the `System.Random` class to fetch the "pseudo-random numbers" in this lesson. We used a LINQ method to count the number of elements in an array that satisfies a certain criterion. We then used the switch expression to select a particular one from a list of candidate expressions. We also used the enum types and static constructors.

The sample code of this lesson could be improved in a number of different ways. In particular, if we define a custom type for `Game`, for instance, then we could store some internal states (like the drawn balls, etc.), which in turn could simplify our implementations.

This is a good segway to the Part II, in which we will explore the object oriented programming styles in C# in more depth.

15.3. Exercises

1. Write a similar program in which the player can pick multiple set of numbers.

Author's Note

Request for Review

Congratulations! You just finished the first part of this book. This could have been the most difficult part, depending on where you are coming from. We covered some "advanced topics" like generics and type constraints in this part.

The real fun starts from the second part, Moving Forward, where we cover C#'s support for the object oriented programming (OOP) paradigm.

If you find this book helpful in any way, then please consider leaving an honest review for other readers, who may find this book useful in learning programming, and programming in C#.

Now, let’s move forward! ☺

Review - C# Key Concepts

We covered a lot of ground in Part I. Here's a list of some of the topics we went through. These are meant to be reminders rather than the real summaries.

If any of the concepts listed here are not very clear to you, then you can go back and review the example code and the lessons again. Or, you can refer to other references for more information.

C# Program Structure

Main Method and Top Level Statements

A C# program can be built into an executable or a library. An executable C# program must include one and only `Main()` method, as an entry point to the program.

The "top level statements" can be used in liu of the `Main()` method. The top level statements are automatically converted to a Main method of an implicitly created class by the compiler. The top level static methods are also converted into static methods of that class.

Namespace

We can use short forms of the names from other namespaces using the `using` or `using static` declarations. Namespaces are primarily used in C# to organize "names" into groups and to minimize the chance of name collisions across different C# programs and libraries. As of C# 10.0, certain namespaces like `System` are implicitly imported and we do not need to use explicit qualifications for the names from those namespaces even without using the `using` declarations.

Comments

C# comments are largely ignored by the compiler tools. You can use single line comments with `//` or multiline comments with `/* */`.

C# Built-In Types

String

`string` is a builtin type in C#. A string literal, like "Hello World!", can be assigned to a variable of the `string` type. Stings can be manipulated, or formatted, using the string interpolation expression, or using the `string.Format()` static method, among other things.

The `@` character in the front of a string literal indicates that the string is to be interpreted verbatim.

Char

`char` is a struct type that represents a Unicode UTF-16 character. The `string` type represents text as a sequence of `char` values.

Primitive Numeric Types

C#, and .NET, supports a number of different integer and floating types. Integer literals can be represented in base 2, base 8, base 16 as well as in base 10.

Boolean Type

`bool` is a value type that represents a Boolean value, either `true` or `false`. To perform logical operations with values of the bool type, we use Boolean logical operators. The default value of the `bool` type is `false`.

Type Conversions

There are a number of ways to convert a value/variable of one type to another. All numeric types include the `Parse()` and `TryParse()` static methods to convert strings to integral or floating point numbers.

Array

An array data type is used to store multiple variables of the same type. Array elements can be of any type, including an array type. An array can be single-dimensional, multidimensional, or jagged.

Custom Types

Enum Type

We introduced the `enum` types, which are user-defined integer types. An enum type ise used to define a set of related constants. The enum values can be cast into integers and vice versa.

Static Class

A `static class` is used to define `static methods`. Static methods are like " functions" in other programming languages.

Access Modifiers

Various access modifiers like `public` and `internal` can be used to control the access levels for your types and their members. The top-level (custom) types can only be either `internal` (they can only be used within the same assembly) or `public` (they can be potentially used by other assemblies).

Properties

The internal data of a variable can be accessed via properties. Properties in C# are like data fields, in terms of syntax, but they are semantically methods, essentially getters and setters.

Methods

The definitions of static or instance methods can be included in a "block", using a pair of curly braces, or they can be stated as an "expression body", using the fat arrow syntax, if the implementation can be represented by a single expression.

Method overloading

We introduced the concept of method overloading. In C#, one can use the same method names for different (but, related) methods as long as they have different argument lists.

Extension Methods

The extension methods are used to "add" methods to existing types. They are static methods, but they’re called as if they were instance methods on the extended type.

Delegates

A delegate in C# is a type, and it is used to represent a class of methods (static methods or instance methods) that have a certain function signature.

Delegate instances can be used just like any other object, and they can be "called" as if they are methods.

Interfaces

We looked at a few important interfaces in .NET, such as `IEnumerable<T>` and `IEnumerator<T>`, and `IDisposable`. An interface type defines "behaviors" (and, possibly their default implementations). We will come back to interfaces in the next part.

.NET Types

Tuples

C# provides a very convenient syntax for values and variables of the `System.ValueTuple` type. The tuples can be used, for example, to return multiple values from a method (syntactically). We also used deconstructions in a number of examples.

DateTime and TimeSpan

`DateTime` and `TimeSpan` are a few of the most commonly used types in C# programs.

System.Random

We used an instance of the `System.Random` class to fetch random numbers (integers of floating point numbers). A `Random` variable needs to be "seeded", and it generates a sequence of "pseudo-random numbers".

.NET Collection Types

We used a couple of .NET collection types, including `List<T>` and `Dictionary<K,V>`. These are reference types just like arrays.

A list stores a list of items (objects), an a dictionary stores key-value pairs.

Yield Return

We introduced a special syntax in C# to create a method that returns an `IEnumerable<T>`, using the `yield return` statement.

Input and Output

The `System.Console` static class includes many static methods for input and output handling via CLI. We used `Console.Write()` and `Console.WriteLine()` for output as well as `Console.ReadLine()` for input.

Part II: Moving Forward

The noblest pleasure is the joy of understanding. -Leonardo da Vinci

The OOP is overrated.

There, we said it. 😂 The OOP, object oriented programming, is not the goal. It is only a tool that can help us achieve our goal.

In general, the software, especially the large scale software, is an extremely complex system. With thousands of interconnected "parts". Try comparing a large software system such as Microsoft Windows operating system, for example, with the most advanced automobile in the world. There is really no comparison. A large software system (that does useful things for us) is very hard to build, and maintain. Software has bugs. Imperfections. All kinds of errors.

In software engineering, "managing complexity" is the top priority in any projects.

When C++ came out in the late 80s and the early 90s (which was based on the earlier OOP ideas), which *could* potentially replace C, everybody was so excited. *The OOP is going to solve all our problems.*

Well, it didn’t. The OOP is still a good programming paradigm, which can help us manage complexity, but it is not a panacea. It is certainly not our goal. We do not do OOP for the sake of OOP. There seem to be some misconceptions about the OOP, especially, among the beginning programmers.

At some point, in our short history, the OOP was so prevalent in the software industry that the languages like Java, and (the early) C#, were invented, which "forced" the object oriented programming styles on the programmers. In these languages, you couldn’t even write a single line of code without following a certain

"OOP style", at least syntactically.

But, those days are over.

When you use C#, you cannot escape from the OOP. It’s built into the language. But, as briefly mentioned earlier, the emphasis has been gradually shifting in recent years.

We focus more on what will make us more productive. How can we produce more/better software with less time and effort? How can we reduce the frequency of having critical bugs? How can we make our code "more readable" so that it can be more easily updated and maintained in the long run? And so forth. The OOP is just *one of the many tools* that we use in software engineering. In fact, C# has been adopting many functional programming styles recently, which can help us write more robust software.

Having said that, you will still have to know what the object-oriented programming is, how to use it, and when to use it, if you want to program in C#.

Now, let’s get started. ☺

Chapter 16. Caesar Cipher

16.1. Introduction

We explored a couple of ways to create custom types in C# in the first part of this book, namely, by using `delegate` and `enum`.

In this second part, we will look at a few more common ways to create a type, in particular, by using `structs`, `classes`, and `records`, as well as `interfaces`. These are the most important components in the C# programming language. When you write a C# program, you will most likely end up spending most of the time creating custom types using these constructs.

As an introduction to these important concepts, we will create a simple type, `Caesar.Cipher`, which implements a Caesarean cipher method. Here's the link to a Wikipedia article: Caesar Cipher [https://en.wikipedia.org/wiki/Caesar_cipher].

16.2. Code Review - Caesarean Encryption and Decryption

A Caesar Cipher is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed distance in the alphabet sequence. For example, with a left shift of 3, `A` would be replaced by `D`, `B` would become E, and so forth.

The shift of 13 is most commonly used and it is known as "rot13". In Rot13, "encryption" and "decryption" are identical operations for a set of English alphabets (with 26 letters).

In the rot13 scheme, the capital letters map as follows, for instance:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
N O P Q R S T U V W X Y Z A B C D E F G H I J K L M
```

Here’s the main program that demonstrates the use of our custom type `Cipher`.

caesar-cipher/Program.cs (lines 3-12)

```
var cipher = new Cipher(10);

var text = "Hello Programming Artistz!";
Console.WriteLine($"plaintext = {text}");

var encrypted = cipher.Encrypt(text);
Console.WriteLine($"encrypted = {encrypted}");

var decrypted = cipher.Decrypt(encrypted);
Console.WriteLine($"decrypted = {decrypted}");
```

It first creates an instance of a type `Cipher` using the `new` operator, and it assigns its reference to a new variable, `cipher`. It calls the `Encrypt()` and `Decrypt()` methods on this variable, and prints out the results.

> The full example code is available at the end of the book, [appendix-code-listing-part2]. As stated, in all sample code in this book, we use the implicit namespace imports for the `System` and other system namespaces (C# 10.0). As it should be familiar by now, the `Console` static class is defined in the `System` namespace.

The type `Cipher` is defined as follows:

caesar-cipher/Cipher.cs (lines 4-34)

```
public sealed class Cipher {
    public int Shift { get; init; }

    public Cipher(int shift = 13) => Shift = Mod(shift);

    public string Encrypt(string clearText) => Rotate(clearText,
Shift);

    public string Decrypt(string cipherText) => Rotate(cipherText,
-Shift);

    private static int Mod(int shift) => (shift % 26 + 26) % 26;

    private static string Rotate(string text, int shift) {
        shift = Mod(shift);

        var sb = new StringBuilder();
        foreach (var c in text) {
            if (char.IsLetter(c)) {
                if (char.IsUpper(c)) {
                    var u = (char)((((c - 'A' + shift) % 26) +
'A');
                    sb.Append(u);
                } else {
                    var l = (char)((((c - 'a' + shift) % 26) +
'a');
                    sb.Append(l);
                }
            } else {
                sb.Append(c);
            }
        }
        return sb.ToString();
    }
}
```

The Cipher class is defined in the Caesar namespace. You can refer to the full code sample at the end of the book, if necessary. We declare namespaces, throughout the code samples of this book, using the file-scoped namespace syntax of C# 10.0, as explained in Part I.

16.3. Pair Programming - User-Defined Types, Reference Types, Constructors, Object Initializers

In this example, class is used to create a custom reference type. Cipher is a reference type. cipher is a reference variable.

In this small example, there is little difference, but in general, it is an important design decision to make, e.g., what kind of types are best for a given object or concept.

We use `class` or `record (class)` to create a reference type, and `struct` or `record struct` to create a value type.

A "type" defines a "template" or "mold" for a variable, if you will, in terms of *data* and *behavior*.

As a rule of thumb, it’s best to use a `struct` value type, or `record struct` for the things that are primarily "data". And, to use a `class` reference type, or `record class`, for the things that are primarily "behavior" (that can also include data).

The `record`, or `record class`, type prescribes an (immutable) reference type that follows value semantics. `records` are generally used to create data types, just like `struct` and `record struct`.

It may sound all rather confusing, if you are new to programming. But no worries. We will go over this subject, again and again, throughout this book. ☺

If we run the main program, then we will get the following console output.

```
$ dotnet run

plaintext = Hello Programming Artistz!
encrypted = Rovvy Zbyqbkwwsxq Kbdscdj!
decrypted = Hello Programming Artistz!
```

As you can see, the text that has been encrypted and decrypted again is the same as the original text.

The following syntax defines a (reference) type `Cipher`:

```
class Cipher {
    // ...
}
```

The access modifier `public` makes this class potentially accessible from outside the current assembly. The keyword `sealed` indicates that this class cannot be "inherited", or "subclassed". (We will discuss the inheritance and polymorphism later in the book.)

The following is a "constructor method", or a "constructor" for short.

```
public Cipher(int shift = 13) {
    // ...
}
```

It looks like a method but it has the same name as the class. And, its method declaration syntax does not include a "return type". The constructors of a public class are typically `public`, but they do not have to be.

Constructors are used to "construct", or create, new objects of the given type.

The constructors can be "overloaded" just like any other methods. That is, one can define multiple constructors as long as they have different parameter lists.

In this example, the constructor `Cipher(int shift)` takes one `int` argument, named `shift`. The notation `int shift = 13` indicates that if no argument is given while calling this constructor, then the value of `shift` will be defaulted to `13`. The method parameters like `shift` are called the "optional parameters".

If one defines a class without any constructor, then its "default constructor" (a constructor that does not have any parameters) is automatically created by the compiler.

This is an example of an (explicitly defined) default constructor.

```
public Cipher() {
    Shift = 0;
}
```

The constructors are used to create an instance of that type, using the `new` operator. For instance, any of the following syntaxes can be used to create a value of type `Cipher` (a reference value in this case, or an "object"). The value is then assigned to the variable named `cipher`.

```
Cipher cipher = new();
```

```
Cipher cipher = new(2);
```

```
Cipher cipher = new(shift: 5);
```

In the first statement, a Cipher object is created with the Shift property set to `13` since no explicit value provided for it. In the second statement, the newly created object has `Shift` of 2.

The last example statement uses a "named argument", `shift: 5`. Note the syntax. Note that the name `shift` is the parameter name that is used in the declaration of the constructor method. The named parameters can be used for any methods, not just for the constructors.

We can also use the implicit `var` declaration for `cipher`. In such a case, the constructor should be explicitly specified in the `new` expression. For instance, each of the following three statements is equivalent to the corresponding example above:

```
var cipher = new Cipher();
```

```
var cipher = new Cipher(2);
```

```
var cipher = new Cipher(shift: 5);
```

Note that the last statement in this example also uses a "named argument".

An instance of a `class` or `struct` (or, their `record` variants) can also be created using the "object initializer" syntax. For example,

```
var cipher = new Cipher { Shift = 2 };
```

It specifies the necessary arguments inside a pair of curly braces (`{` and `}`). Note that we can use this syntax because `Cipher` provides a no-argument constructor (via the optional argument syntax).

We could have also done like this:

```
var cipher = new Cipher { };
```

Or, even

```
Cipher cipher = new() { Shift = 2 };
```

Or

```
Cipher cipher = new() { };
```

Using the default value of `13` for the `Shift` property.

In many cases, especially for the types that are mainly "data types", it is typical to do away with the constructors (other than the default constructor) and just use the object initializers.

Therefore, you will more likely see the object initializer syntax for the `struct` and `record struct` types, and to a lesser degree for the `record class` types.

Note that using the object initializers requires that the type should have publicly settable, or at least init-only, properties defined for the necessary data fields.

The following line in the class `Cipher` defines a public property `Shift`. (Line 5.)

```
public int Shift { get; init; }
```

This is called an "auto property" (or, auto-implemented property).

Normally, the data is stored in the (private) "fields", and the (public) properties are used to access the data.

But, in the examples like this, the "backing field" is implicit. The compiler automatically generates the implementations for `get`, and `init`, using the implicit field variable behind the scene.

It is typical for a property to be one of the following four: `{ get; }`, `{ get; set; }`, `{ get; private set; }`, or `{ get; init; }`.

- `{ get; }` is essentially a readonly property.
- `{ get; set; }` is a read-write property.
- `{ get; private set; }` means that the setter is private (regardless of the overall access modifier of the property).
- Finally, a property with `{ get; init; }` is called the init-only property. This is a readonly property, but constructors and initializers can still access the property during the construction/initialization of an instance.

Note that, in the above constructor and initializer examples, we are able to set the value of the property `Shift`. Once an object is created, however, the value of `Shift` cannot change.

For an "auto-property" like this,

```
public string Name { get; set; }
```

The compiler would generate a code like this:

```
private string name;
public string Name {
    get => name;
    set => name = value;
}
```

Here `value` is a special predefined variable to denote the value from the assignment expression in the setters.

In this example, `name` is a "field". Although it is legal to have a public field, the fields are almost always private. We typically use properties if the field data need to be exposed in some way.

You can access a property using the dot notation. For example, The following gets the `Name` (from an object called `person`) and assigns it to a variable `name`:

```
var name = person.Name;
```

You can assign a new name to the `person` object using the setter:

```
person.Name = "new name";
```

In this example, the string literal `"new name"` will be passed in to the setter definition as the special value, `value`.

Unless there is a special requirement for the getters and/or setters for data, using auto properties works just fine and it often leads to a more compact and more readable code.

Of course, the properties do not entirely eliminate the need for the "fields". Some data may be truly private to a class, or specific to a particular implementation, and they may not need an exposure through properties. (Although private auto-properties are allowed, the private property syntax is less commonly used.)

The (private) fields are implementation details, whereas the (public) properties are part of the public APIs for a given type.

> The fields are variables whereas the properties are (essentially) methods in C#.

The `Cipher` type defines two core APIs as instance methods. Namely, `Encrypt()` and `Decrypt()`.

Since their implementation is "symmetric", we introduce a static helper method, `Rotate()`, in implementing both methods. If the encryption is shifting letters *to the left* by `S`, then the decryption is shifting the letters *to the right* by `S`.

Summary

We defined our first custom type using `class` in this lesson. We looked at the constructor and initializer syntaxes for `class` (which are also applicable to `struct`, and `record` and `record struct`, as we will see in the following lessons).

We also discussed some of the essential members of a class, namely, instance fields, instance properties, and instance methods.

> ℹ Note that these are normally called just the fields, properties, and methods, respectively, unless they need to be specifically distinguished from the static fields, static properties, and static methods, which we discussed in the first Part.

Tip - So, What is OOP?

In the beginning of this part, we stated that the importance of the OOP style has been diminishing, in general, and in the "modern C#", in particular.

Still, OOP is one of the most important component of C#. When you develop games in Unity, for example, you will use the OOP. When you program Web services using ASP.NET, for instance, you will use the OOP. When you do Windows programming using the Win32 API, for instance, you will use the OOP. The OOP is everywhere as long as you program in C#.

Hence, it is of paramount importance for you to understand the basic concepts of OOP. The details are less important.

> You can skip this section if you are familiar with the OOP style from other languages like Java or C++. If you know how to, and what it means to, "override" a "virtual method", for example, then you are all set. ☺

So, what exactly is the "object oriented programming style"? Before we discuss what it is, let's look at the problems that OOP is trying to solve.

When a software grows, its complexity increases super-linearly. It is hard to quantify, but in general, we can assume that it grows, or it can possibly grow, exponentially.

Let's suppose that you have a system with `N` components (anything) and they are all connected to each other, not just between the pairs, but between the triplets, … and among all of them. The number of connections grows as `2^N`. In the worst case, the "complexity" (regardless of how we exactly define it) can potentially grow exponentially with the number of its constituent components.

In theory, one can write such bad software that they could behave like this. In practice, nobody can possibly write such bad software. 😀 But, the problem was, in the early days, that there was no way to manage this kind of complexity, at least as long as we used the imperative programming style.

For example, although we do not use this term very much these days, the "global variable" was considered *evil*. A global variable can be accessed by anyone (in the program), and it can be changed by anyone. Think about the exponential complexity. As the program grows, it will get harder and harder to understand the program. Because, essentially, everything is connected to everything.

But, there was no way to prevent this, for instance, in the programs written in the C programming language. We had to rely on the programmers' competence, which did not always work out. ☺

One way to manage this kind of complexity is to use "components" or "modules", broadly speaking. We mentioned this in the software stack metaphor earlier. By "compartmentalizing" big software into smaller constituent components, and managing their "connections", we can reduce the run-away complexity problem. That is the idea.

C# supports, broadly speaking, this kind of component style programming. Whether we view an assembly as a component or a custom type as a component is not that important in our discussion. In theory, one can imagine a hierarchy of different kinds of components in more than one or two levels.

The object oriented programming is, in some sense, a special kind of the component based, or modular, programming style. In the OOP, there is a concept of an "object" (an abstract concept, not necessarily the same as an `System.Object` object of C# and .NET). And the object knows how to manage itself. It hides all its "complexities" (again, we are just using these terms without precisely defining them), and it exposes only what is needed by the "outside world".

Anybody who interacts with an object need not know anything about its internal complexities, e.g., how it implements a certain function, or what kind of data it holds, etc. That is the object's job.

Anybody interacting with the object only needs to know how to use it and what services, if any, they need to provide back to the object. This is generally called the "interface" (again an abstract concept, not necessarily the same as C# `interface`). In the OOP, the "hiding" part is often called the "data encapsulation".

There you have it. Data encapsulation, and the controlled interactions through well-defined interfaces. We build a program using "objects". That is the object oriented programming. Inheritance, polymorphism, etc. are all secondary.

Let's consider the following example, not specific to any particular programming language.

Let's suppose that we want to know the length of an "array" variable, that is, the number of elements in the array. One way to do this is to use a (global) function `length()` or something like that. For instance (in pseudo-code),

```
arr = new array();
len = length(arr);
```

Let's consider an alternative. For instance (again, in pseudo-code),

```
arr = new array();
len = arr.length();
```

The second example is more like an OOP style than the first. Why? It's not just the syntactic difference.

In the second example, the variable `arr`, and only this variable, knows the count of its elements. And, we need to ask `arr` for its length. The `arr` variable in this example is more like an "object". On the other hand, in the first example, what is the `length()` function? How does it know how many elements does the `arr` array have? This example is less like an OOP style.

Of course, this is the broadest conceptual definition of the OOP. Many people will disagree on what exactly the OOP is. But, as stated, the details are not that important. It is the core concept that really matters.

Different programming languages support the OOP, if they do, in different ways. In C#, everything is an "object". At least, at the syntactic level. That is why we call it *an object oriented programming language.* ☺

> As stated, the "static methods" are more like functions, and the fact that we are defining them in a class (or struct, interface, ...), not globally, is really a "hack" in the "pure OOP" languages like Java and C#. Not everything can fit into the OOP style. ☺
>
> Nevertheless, the static methods provide certain benefits, say, compared to the globally defined functions. For one, the static variables (of a static class), which a static method can access and manipulate, are not "global variables". (At least, technically speaking, although in practice, both static and instance variables of a large class/object are just as bad as "global variables".) The static methods still provide a certain level of "data hiding".

Chapter 17. Rational Number

17.1. Introduction

We used `class` to define a type in the previous lesson, namely, `Caesar.Cipher`. Let's try to define a new type using `struct` in this lesson.

17.2. Code Review

The rational number is a number that can be expressed as a fraction of two integers, a numerator and a non-zero denominator.

For example, `0.6` is a rational number since it can be represented as `3/5`. On the other hand, `π` is not a rational number since there is no such representation. If you need a refresher, here’s the link to a Wikipedia article: Rational Number [https://en.wikipedia.org/wiki/Rational_number].

We define a value type `Number.Rational` using `struct` in this lesson. The main program demonstrates a few usages of the values of the `Rational` type.

rational-number/Program.cs (lines 3-17)

```
var a = new Rational(4, 8);
var b = new Rational(9, 12, true);
Console.WriteLine($"a = {a}; b = {b}");

var sum = a + b;
Console.WriteLine($"Sum =\t{sum}");

var sub = a - b;
Console.WriteLine($"Sub =\t{sub}");

var prod = a * b;
Console.WriteLine($"Prod =\t{prod}");

var div = a / b;
Console.WriteLine($"Div =\t{div}");
```

rational-number/Program.cs (lines 19-26)

```
var c = new Rational(1, 2);
Console.WriteLine($"c = {c}");

if (a.Equals(c)) {
    Console.WriteLine("a == c");
} else {
    Console.WriteLine("a != c");
}
```

The type `Rational` is defined as a public "readonly struct":

rational-number/Rational.cs (lines 3-24)

```
public readonly struct Rational {
    private readonly ulong num, den;
    private readonly bool neg;

    public Rational()
        : this(0, 1) { }

    public Rational(ulong numer, ulong denom)
        : this(numer, denom, false) { }

    public Rational(ulong numer, ulong denom, bool negat) {
        if (denom == 0) {
            throw new ArgumentException("Zero denominator");
        }

        if (numer == 0) {
            (num, den, neg) = (0, 1, false);
        } else {
            var cf = GCD(numer, denom);
            (num, den, neg) = (numer / cf, denom / cf, negat);
        }
    }
```

"Overloading" unary operators `+` and `-`:

rational-number/Rational.cs (lines 26-28)

```
26      public static Rational operator +(Rational a) => a;
27      public static Rational operator -(Rational a) =>
28          new(a.num, a.den, !a.neg);
```

"Overloading" binary operators `+`, `-`, `*`, and `/`:

rational-number/Rational.cs (lines 30-50)

```
30      public static Rational operator +(Rational a, Rational b) {
31          var cf = GCD(a.den, b.den);
32          var (f1, f2) = (a.den / cf, b.den / cf);
33          var denom = cf * f1 * f2;
34
35          ulong numer;
36          bool negat;
37          if (a.neg == b.neg) {
38              numer = a.num * f2 + b.num * f1;
39              negat = a.neg;
40          } else {
41              var (n1, n2) = (a.num * f2, b.num * f1);
42              (numer, negat) = (n1, n2) switch {
43                  var (x, y) when x == y => (0ul, false),
44                  var (x, y) when x > y => (n1 - n2, a.neg),
45                  _ => (n2 - n1, b.neg),
46              };
47          }
48
49          return new(numer, denom, negat);
50      }
```

rational-number/Rational.cs (lines 52-53)

```
    public static Rational operator -(Rational a, Rational b) =>
        a + (-b);
```

rational-number/Rational.cs (lines 55-56)

```
    public static Rational operator *(Rational a, Rational b) =>
        new(a.num * b.num, a.den * b.den, a.neg ^ b.neg);
```

rational-number/Rational.cs (lines 58-61)

```
    public static Rational operator /(Rational a, Rational b) =>
        (b.num != 0)
            ? new(a.num * b.den, a.den * b.num, a.neg ^ b.neg)
            : throw new DivideByZeroException();
```

The `object.ToString()` method is "overridden":

rational-number/Rational.cs (line 63-64)

```
    public override string? ToString() =>
        $"{(neg ? "-" : "")}{num}/{den}";
```

A `private` helper static method that implements the "Euclid algorithm" to compute the greatest common divisor (GCD):

rational-number/Rational.cs (lines 66-71)

```
    private static ulong GCD(ulong a, ulong b) {
        while (b > 0) {
            (a, b) = (b, a % b);
        }
        return a;
    }
```

17.3. Pair Programming - Value Types, Method Overriding, Operator Overloading

All value types, both built-in value types and custom struct and enum types, inherit from `System.ValueType`, and ultimately from `System.Object`.

As stated, C# (and .NET) uses a "unified type system". Every type is a child of `object` in C#, or `System.Object` of .NET. To put it differently, every type in C# is "derived", either directly or indirectly, from `object`.

A type `A` *deriving* from another type `B` includes, or "inherits", all the "behaviors" of its parent type `B`. And, the child type `A` can possibly have other "behaviors" as well that are not present in its parent type.

In C#, the "behaviors" of a type is represented by a set of public properties and methods.

A type defined with `class`, `struct`, `record`, or `record struct` in C# can also include, and use, certain internal data, e.g., the `public` or `protected` fields, which are defined in its parent type(s).

A type defined with `class`, `struct`, `record`, `record struct`, or `interface` in C# can also use certain implementations of its parents, that is, their `public` or `protected` properties and methods.

Note that `public` methods and properties can be both "behaviors" and "implementations". Although a child, or derived, type cannot overwrite, or change, the behavior or the contract (e.g., API), it can "override" the parents' implementations, if allowed.

A non-`sealed` class or record class can mark certain properties or methods overrideable by any child class or child record class, by using the `virtual` modifier in front of their definitions. The implementations of non-virtual properties and methods cannot be overridden.

An `abstract` type (class or record class) can also declare that certain properties or methods *must* be overridden by its child type by using the `abstract` modifier in front of their declarations. An abstract property or method declaration cannot include its definition.

Or, to put it differently, a property or method that does not have a definition must be marked as `abstract`. A type that includes at least one abstract method or property must be declared as an `abstract` type (class or record class). An `abstract` type, however, need not include any abstract properties or abstract methods. An `abstract` class or `abstract` record cannot be instantiated regardless of whether it includes any abstract properties or methods.

All public methods and properties in an `interface` are *implicitly* either `virtual` or `abstract`, depending on whether they have any default implementations or not, and they can always be overridden. The `abstract` methods in an interface must be implemented in its (non-abstract) child classes, structs, or records.

Just like an abstract class (or record), an interface cannot be directly instantiated.

> Now that we can include (default) implementations in an interface, there is really little difference between an abstract class and an interface.
>
> The main, and essential, difference is that an interface cannot include data (e.g., "fields"). Otherwise, in most cases, it is up to the programmer which one to choose as a base for other custom types. In general, if it’s about the "pure behavior", then the

> `interface` is preferred.
>
> One other thing to note is that C# only supports the "single inheritance" when it comes to the concrete base types. That is, a concrete type cannot have more than one concrete type parent. It should also be noted that a `struct` can implement an `interface(s)` while it cannot inherit from other `structs`. Hence, broadly speaking, interfaces are generally preferred.

A non-`interface` child type `X` that inherits from an `interface` `I` can provide implementations for the properties and methods of its parent interface `I`. And a concrete child type `X` must do so, if no default implementations are provided in `I`, or in any of its ancestors. In such a case, we say that the concrete type `X` "implements" an interface `I`. (That is, rather than using the more general term "inheritance".)

As just stated, a class can inherit from no more than one (non-sealed) parent class. Likewise, a record class can inherit from no more than one (non-sealed) parent record class. A class, struct, or record type (record class and record struct), as well as an interface type, can inherit from zero, one, or more interface types.

> We can still use the term "multiple inheritance" in case where a concrete type implements more than one interfaces, just like when an interface inherits from multiple parent interfaces. As stated before, not everyone may agree with the terminologies, but it is the concept that is important and not the words that we use.

A type that inherits from one type can also be inherited by another. In theory, there is no limit. A parent-child hierarchy can be as long, or as deep, as you want. In practice, however, a very large type hierarchy is pretty rare. (As alluded before, the inheritance has some performance implications.)

The only constraint in C#/.NET is that every class or record class type should inherit, either directly or indirectly, from `System.Object`.

As stated, `enum` and `struct` (and `record struct`) types (implicitly) inherit from `System.Enum` and `System.ValueType`, respectively. `System.Enum` inherits from `System.ValueType`, which in turn inherits from `System.Object`.

An `enum` type cannot be inherited from any other type than `System.Enum`. A struct type, including a record struct type, can only be inherited from `interfaces` other than `System.ValueTypes`.

An object of a derived type can be used in the places where an object of its parent type is expected. This is called the "polymorphism". (Or, the "interface-based polymorphism".)

In particular, a reference variable of a type `X` (class, record, or interface) can be used in place of a reference of a type `Y` as long as `X` inherits from `Y`, either directly or indirectly.

In case of `struct` or `record struct`, a value type, we cannot directly use the polymorphism. But, it can be done using the interfaces that a struct type implements, which requires "boxing" and "unboxing". More on this later.

As stated earlier, we can only use the polymorphic behavior for reference types since the polymorphism requires some kind of "indirection", and a reference, by definition, "points to" its underlying data, which is a one-level indirection.

> This is a lot to take in, if you are new to C#. But, these "rules" will seem natural after getting some experiences in programming in C#. You do not have to "memorize" them at this point.

The (ultimate) base type, `System.Object`, defines the following three virtual methods, among other things,

```
public virtual bool Equals (object? obj);
public virtual int GetHashCode ();
public virtual string? ToString ();
```

These methods are defined, and available, in all types, whether builtin or user-defined, since every single type in .NET inherits from `System.Object`, or `object` in C#. In other words, these "behaviors" are included in all types in C#/.NET.

Note that these three methods are declared as `virtual` (but, not as `abstract`), indicating that they provide certain default implementations, which may or may not be entirely appropriate for the particular derived types.

`System.ValueType` "overrides" the following two methods, among other things, to provide a value type with the default "value semantics".

This is done using the `override` modifier in C#.

```
public override bool Equals (object? obj);
public override int GetHashCode ();
```

Sometimes, you may have to override these methods in your own value types as well. `Number.Rational`, a value type defined with `struct`, just uses the default implementations from `System.ValueType`.

The `System.ValueType` type overrides the `System.Object.Equals()` virtual method to provide "value semantics" to its child `enum`, `struct`, and `record struct` types. When you override the `Object.Equals()` method, you will need to override the `Object.GetHashCode()` method as well.

We are not going to go into details about this "requirement" in this book, but normally, two objects of the same "hash" value should

> be equal to each other, and vice versa. Hence you will need to override both methods to provide consistency implementations for your custom type.
>
> Otherwise, the objects of your custom type can show many strange, and seemingly erratic and inconsistent, behaviors. ☺ (In other words, your custom type will not be generally usable, across broad use cases.)

As with a type created with `class`, a value of `struct type` can be instantiated using the constructors or object initializers.

In this particular case of `Rational` we do not use any properties, and hence the object initializer syntax has limited use.

We create three variable, `a`, `b`, and `c`, using the overloaded constructors.

```
var a = new Rational(4, 8);
var b = new Rational(9, 12, true);
var c = new Rational(1, 2);
```

Then, the program tests the four basic arithmetic operations between pairs of `Rational` numbers, addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). Finally, it calls `Rational.Equals()` to verify the value semantics for these variables.

```
$ dotnet run

a =  1/2; b = -3/4
Sum =   -1/4
Sub =    5/4
Prod =  -3/8
```

```
Div =   -2/3
c =  1/2
a == c
```

We define a new `struct` type as follows:

```
struct Rational {
    // ...
}
```

The `Rational` struct is declared as `public`.

In addition, it also includes a modifier `readonly`. A properly defined `readonly struct` creates an "immutable" type. The value of an immutable variable cannot change once created/initialized.

It is a good practice to make a value type immutable. The "identity" of a value is determined by its value, *be definition*. It is cumbersome, difficult, and error-prone to deal with the values which do not have *fixed values*. ☺

> As we will discuss throughout this book, a truly "readonly struct" should not contain any non-immutable reference type fields.

In this example, we define `Rational` with three values, `num` for the numerator, `den` for the denominator, and `neg` for the sign. (Lines 4-5.)

```
private readonly ulong num, den;
private readonly bool neg;
```

Note that these fields are all declared `readonly`. And, as stated, we do not expose them externally via any public properties.

A value type should have a "default value". The default value of a nullable reference type is `null`.

The primitive types all have default values. For example, the default value of all integral types is zero (`0`), and that of the `float` types, `float` or `double`, is zero as well (`0.0`). The default value of `bool` is `false`.

The (implicit) default constructor of a struct type typically creates the default value of the type. The default constructor initializes all its fields with their respective default values. Unlike in the case of `class`, the default constructor is always created (by the compiler), unless it is overridden, regardless of whether the `struct` has other constructors or not.

> Note that, starting from C# 10.0, the parameterless default constructor of a `struct`, and a `record struct`, can be overridden. They need to be `public`, however, unlike in the case of the `classes`.

In the case of `Rational` in our example, the implicit (compiler-generated) default constructor sets `num` and `den` to `0` and `neg` to `false`, by default. In this particular case, however, that is not a valid value for the `Rational` type. `den` cannot be zero.

If we use the default constructor (without overriding it),

```
var d = new Rational();
Console.WriteLine($"default = {d}");
```

It prints out `0/0`, which is an invalid value. Likewise, the default operator `default(Rational)` would produce the same invalid value `0/0`.

Although we do not use the zero value in this particular illustration, we should therefore define our own default value for the type by overriding the default constructor (lines 7-8).

```
public Rational() : this(0, 1) { }
```

This `this(0, 1)` syntax calls the particular overloaded constructor with the two parameters, lines 10-11, as we will further discuss shortly. In the context where the default value of the type `Rational` is needed, we can use the `default` value, which now refers to `0/1`. As stated, it is recommended to ensure the correct default value for all value types.

It should be noted that you cannot overload the `default` operator in C#. Hence you should override the parameterless constructor if the default implementation does not suit your needs.

Alternatively, one can define a special value, which can be used as the default value. For example,

```
public static readonly Rational Zero = new(0, 1);
```

This way, we can explicitly use `Rational.Zero` rather than the `default` default. However, now that C# 10.0 allows overriding the default constructor (which takes no parameters), that is the preferred way to achieve the same.

We define three constructors for `Rational`. As mentioned, the parameterless constructor, lines 7-8, overrides the compiler generated default constructor.

Note the syntax where one constructor calls another constructor. For example, in lines 10-11,

```
public Rational(ulong numerator, ulong denominator)
```

```
    : this(numerator, denominator, false) { }
```

The keyword `this` is used after the colon ":" in the constructor declaration. In this case, `this(…)` calls the three parameter constructor.

We could have achieved more or less the same thing without using the two parameter constructor, in this particular example, by using the optional argument syntax:

```
public Rational(ulong numerator, ulong denominator, bool negative =
false) { /* ... */ }
```

> People seem to have different opinions as to whether it is a good practice to throw an exception in a constructor, especially in the languages like Java and C#. The author does not see any real/practical problems by throwing an exception in a constructor, and we do that in the `Rational` constructors, e.g., lines 14-16, when an invalid denominator is used. Often, "fail fast" is the cleaner and easiest solution. Alternatively, we could have used different approaches, like making the constructors private and using a factory method to create new instances of a type, etc. This discussion is, however, beyond the scope of this book.

One thing to note is that we "normalize" the internal representation of a rational number in the constructors. That is, the number `0` uses the unique canonical representation, `0/1`, although all other representations with `0` in the numerator such as `0/2`, `0/3`, etc. would all equivalently represent `0`. We do the same with non-zero `Rational` numbers, using the `GCD` static function. In this implementation, `num` and `den` are "mutually prime" to each other. That is, one is not integrally divisible by the other.

We will leave it as an exercise to the readers to figure out how this implementation yields a unique representation for each different rational number.

C# supports the "operator overloading" for a certain set of operators.

The `Rational` type overloads two unary operators and four arithmetic binary operators. Note their syntax. The methods overloading any predefined C# operators should be both `public` and `static`.

> Note that the "limitation" of the current generics, as we explained in Sort Numbers, is due to the fact that the operators are `static` methods in C#. As mentioned, this particular problem will be solved, likely, in C# 11.0, when the *static abstract methods* are allowed in `interfaces`.

Note that we tend to use a lot of expression bodies in the sample code. For example, the division (`/`) operator could have been written as follows using the block body:

```
public static Rational operator /(Rational a, Rational b) {
    if (b.num == 0) {
        throw new DivideByZeroException();
    }
    return new(a.num * b.den, a.den * b.num, a.neg ^ b.neg);
}
```

It is common to overload the equality (`==`) and inequality (`!=`) operators for value types. In order to do that, however, we also need to override both `Equals()` and `GetHashCode()` methods.

For instance,

```
public static bool operator ==(Rational a, Rational b) =>
```

```
a.Equals(b);
public static bool operator !=(Rational a, Rational b) =>
!a.Equals(b);
public override bool Equals(object? obj) =>
    obj is Rational rat && neg == rat.neg && num == rat.num && den ==
rat.den;
public override int GetHashCode() => HashCode.Combine(neg, num, den);
```

For many value types, providing custom "conversion" or "cast" operators can be useful as well. There are `explicit` and `implicit` conversions.

Here are a couple of examples:

```
public static explicit operator double(Rational r) => (double)(r.neg
? 1.0 : -1.0) * ((double)r.num / r.den);
public static implicit operator Rational(long n) => new((ulong)((n <
0) ? -n : n), 1, n < 0);
```

Summary

We reviewed how to create a custom type using `struct` in this lesson. True value types should be created via `readonly struct` and they should not normally contain fields of reference types such as arrays and what not.

(As we will see shortly, however, records are a special kind of reference types.)

Tip - Value vs Reference

We have been using the term *"value"* throughout this book without precisely defining what exactly it is.

It is actually difficult to give a precise definition. A `value` is what the computer program deals with. Values can be stored in memory, e.g., during the execution of a program. A number `3` is a value, as well as a string "Hello".

When a function returns to the caller with a value, the value is copied, e.g., from one stack frame to another, or, more precisely, from the callee frame to the caller frame.

There are a special kind of values, namely, "references". A reference is special in that it "references" another value. Although it itself is a value (like number `3`, which can be copied, etc.), we do not mostly care about its own value in the program. We primarily care about the value that it references (or, "points to"). We often use the term "value" for non-reference values only. We use the terms *value* and *reference* almost mutually exclusively (although that is not entirely true).

Now, it is rather important to understand these concepts to be able to use C# more productively and effectively.

Sometimes, these concepts are tied to other related concepts like the stack memory and the heap memory. We will, however, mostly focus on the conceptual framework in this section rather than going over all the practical aspects. In theory, values and references can be stored in any kind of memory. The data that a reference points to, however, needs to be stored in a location in a "permanent", or at least "semi-permanent", memory (e.g., the heap).

A program construct, `variable`, can be used to store a value in a program. A variable has a "type". A type defines what kind of values a particular variable can have. And, what kind of operations are allowed on that variable, and so forth.

For example, a (mutable) variable of the `int` type can be used to store any valid 32 bit signed integer numbers like `1000` or `-25`. It can also be used in an addition operation, etc. But, it cannot be used to store a string value like `"hi"`, or one cannot call a method like `Run()` on this variable since the `int` type does not have such a method defined.

For statically typed languages like C#, the precise types of (most of) the variables in a program are determined at the build time. There are some exceptions though. For instance, "polymorphism" is a run time behavior, as we will discuss later in the book. C# also has the `dynamic` keyword, which allows bypassing the compile time type checking.

As a corollary to the fact that a variable is associated with one fixed type, a variable can store either a value or a reference, but not both, in C#. That is, a variable is either a reference variable or a value variable.

In programming languages like C/C++, Go, and Rust, for each value type there is generally a corresponding reference type, that is, a "pointer type". Or, a custom type can be created, in languages like C++, that can behave like a reference type or a value type. A single type can behave like both value and reference types.

C# works differently. (As mentioned, we do not discuss the "unsafe" C# in this book.) In C#, a type is either a value type or a reference type. A variable of a value type can only store values (of that type) whereas a variable of a reference type can only store references (of that type).

> In many C-descendent languages, including Java and Javascript, virtually all types are reference types. These languages are much simpler, and easier to use (by sacrificing flexibility), and that can be one of the reasons why these language are more popular than C++ or C#.
>
> As alluded in the introduction, C# is an advanced, and very powerful, programming language, and it is not for the faint-of-

> heart. ☺

The "modern C#" now has a number of features that can make values behave like references, and vice versa. We saw some examples in earlier lessons, like the keywords, `ref`, `out`, and `in`.

Values are typically "copied". (Or, in general, they can also be "moved".) That is, once a value is copied, we end up with two distinct values (with the same (initial) value). On the other hand, when a reference is copied, both the source and target references still point to, or reference, the same "value" (data).

As stated, a reference is a value, and the references are indeed copied. But, that is almost irrelevant in the program. For the reference variables, we mainly care about the underlying value/data which a reference points to. And they are one and the same "value" for both source and target reference variables after the "copy".

Let’s take a look at a few examples. An `int` in C# is a value type.

```
var a = 10;
ChangeToTwenty(a);
Console.WriteLine($"a = {a}");

static void ChangeToTwenty(int a) {
    a = 20;
}
```

In this example, the program prints out `a = 10` to the console. Calling the method `ChangeToTwenty()` did "nothing" to the var `a` (in the main part of the program).

When the method is called, the value of `a` is "copied" to the argument `a` of the method. Now they are two distinct values. Changing the value of `a` inside the method has no effect to the (separate) value `a` outside the method. This is called " value semantics". Values are "copied".

If we use a reference value, then the result will be different. An array, e.g., `int[]`, in C# is a reference type.

```
var b = new int[] { 1, 2 };
AddOneToElements(b);
Console.WriteLine($"b = {{{string.Join(",", b)}}}");

static void AddOneToElements(int[] x) {
    for (var i = 0; i < x.Length; i++) {
        x[i]++;
    }
}
```

If we run this program, then its output will be `b = {2,3}`.

Unlike in the previous example, changing the values of `x` (of a type `int[]`) inside this method, changed the values of the outside variable `b`. This is called "reference semantics".

When we call the method, `AddOneToElements()`, the value of `b`, a reference, is indeed copied. Inside the method, the `x` is different than the `b` argument of the method. But, the important thing is that *both point to the same underlying value,* `{1,2}`. When we changed the underlying value/data of the reference `x`, it changed the underlying value/data of the outside variable `b` because they point to the same value/data.

An (underlying) value can be shared by multiple references, and changing (the underlying value of) one reference can affect (the underlying values of) all other references.

In languages that support pointers, the "value" of a pointer/reference is the address of the memory location where the underlying value is stored. C# uses something similar although it may or may not be exactly the memory address, in general, since

C# is a garbage-collected language.

But that fact is only secondary. "Reference semantics" is conceptually what we just described here regardless of how it is actually implemented in a particular programming language/runtime.

> C# does have pointers (as well as keywords like `fixed`), which can be used in an "unsafe" context. As stated, we do not discuss the "unsafe C#"" in this book.

The underlying values of reference variables are generally stored in the "heap memory". The heap memory is allocated to a running program, and it remains until the program terminates. To support the reference semantics, as stated, the underlying values need to be stored in some "permanent" locations.

On the other hand, the values of value types or reference values (not the underlying values) can be, and typically are, stored in transient place known as the "stack memory". When a function/method is invoked, a new stack frame is created, and all temporary, or "local", variables are stored on the stack. When the function/method returns, the stack frame is destroyed and all values stored in that frame are lost (unless they are "copied" to somewhere else).

One of the most important differences of these memory locations is that allocating memory on the heap is generally more expensive because it has some overhead of managing memory. In contrast, creating and destroying values on the stack memory is much cheaper. In this sense, `values` of the value types, which are almost always stored on the stack, are preferred.

On the flip side, copying large values can be expensive. References are small values (e.g., on the order of a few bytes depending on the machine architecture), and copying their values (not the underlying values) can be rather efficient. It is typically comparable to copying integer values. Hence, in certain situations, reference variables are preferred over value variables.

We are ignoring many details, but that's the most important takeaway from this lesson. Values can be useful, and the value semantics can be useful, in certain situations. References can be useful, and the reference semantics can be useful, in certain situations.

As a programmer, you will have to figure out what to use when, when you have a choice.

> As stated, in many programming languages like Java or Python, you do not have much choice. The choice in C# is the power. And, as you know, *with great power comes great responsibility.* ☺

Unlike in languages that support pointers, normally, in C#, the values of a value type follow value semantics, and the values of a reference type, or references, follow reference semantics.

C# provides a way to use reference semantics for values in certain situations. The `ref` keyword (and its two siblings) can be used when a value is passed to a method as an argument, or when a value is returned from a method. Here's an example:

```
var c = 10;
MultiplyByTwo(ref c);
Console.WriteLine($"c = {c}");

static void MultiplyByTwo(ref int c) {
    c *= 2;
}
```

The output of this program is `c = 20`.

It is as if a (one-element) array has been created that contains `c`, and that array is used to call `MultiplyByTwo()`, as in the previous example using `AddOneToElements()`. That is reference semantics. Although `int` is a value type, and `c` in this case is a value, C# allows us to use reference semantics when needed/desired in certain situations. (This is sometimes called "call by reference" as opposed to "call by value".)

> Note that C#'s `ref return` effectively provides "move semantics". As stated, copying (large) values can be expensive, and `in`, `ref`, and `out` essentially provides a workaround.

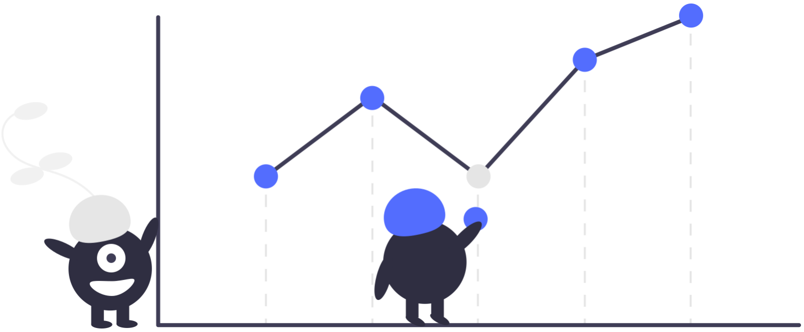
On the flip side, if you want to use value semantics for references, then the convention, in the languages like Java and C#, is to override, and use, methods like " Object.Clone()" or "Object.MemberwiseClone()" to implement the "deep copy" logic, say, instead of using the assignment operator (`=`), which does `shallow copy`.

> In fact, C# allows "operator overloading", as we will discuss in the next few lessons. Hence, we can create a custom type using `class` (which by default follows reference semantics) and make it behave like a value type. As we will see shortly, that is what `record` (introduced in C# 9.0) does for us without us having to implement the value logic through operator overloading and what not.

Chapter 18. Stock Ticker

18.1. Introduction

We used `class` and `struct` to create custom types in the previous two lessons. Let’s try using `record` this time.



The `record` type was first introduced in C# 9.0, and it is now can be used with both `classes` (reference types) and `structs` (value types) (as of C# 10.0). And, depending on what kind of programming you generally do, `records` can be one of the best things that ever happened to the C# programmers.

One of the worst things in programming is doing the same things again and again. For example, we often need to create a number of types that are, in many ways, very similar to each other and yet different enough to be a single type. We end up with a lot of similar code, much of which may be called the "boilerplate" code.

C#'s `record` is one of the efforts to reduce the boilerplate code.

In general, "less code" is a good thing. Less code means less code to write to begin with, less code to read, and less code to maintain.

> Writing less code means that you’ll have more time to do other things. Or, you’ll have more time to do *more coding.* ☺

18.2. Code Review - Fake Stock Ticker Service

We use a "fake" stock ticker app as our sample program in this lesson.

The main driver program checks the prices of (fake) Apple and (fake) Microsoft stocks every half a second (or `500` milliseconds), and if the price has changed for a stock, then it prints out the stock record for the given stock symbol.

stock-ticker/Program.cs (lines 3-22)

```
const int intervalMillis = 500;

var symbols = new[] { Prices.AAPL, Prices.MSFT };

var stocks = new Dictionary<string, Stock>() {
    { Prices.AAPL, new(Prices.AAPL, 0.0m) },
    { Prices.MSFT, new(Prices.MSFT, 0.0m) },
};

while (true) {
    foreach (var symbol in symbols) {
        var stock = Ticker.Get(symbol);
        if (stock != stocks[symbol]) {
            stocks[symbol] = stock;
            Console.WriteLine(stock);
        }
    }

    await Task.Delay(intervalMillis);
}
```

Note that the program runs indefinitely through the `while(true)` loop.

Ticker is a "service" that returns a stock record for a given stock symbol. It updates the stock price by calling the *Prices* service.

stock-ticker/Ticker.cs (lines 3-13)

```
public static class Ticker {
    private static readonly Dictionary<string, Stock> stocks = new() {
        { Prices.AAPL, new(Prices.AAPL, Prices.Stock[Prices.AAPL]) },
        { Prices.MSFT, new(Prices.MSFT, Prices.Stock[Prices.MSFT]) },
    };

    public static Stock Get(string symbol) {
        stocks[symbol] = stocks[symbol].UpdatePrice(Prices.Stock[symbol]);
        return stocks[symbol];
    }
}
```

In this example, `Ticker` is implemented as a static class, and it exposes one API, `Get()`, as a static method.

The (fake) "Prices" service simulates the real-time stock price changes using a series of randomly generated numbers.

stock-ticker/Prices.cs (lines 3-19)

```
sealed class Prices {
    internal const string AAPL = "AAPL";
    internal const string MSFT = "MSFT";

    private static readonly Prices instance = new();
    internal static Prices Stock => instance;

    private Random rand;
    private Dictionary<string, decimal> prices;

    private Prices() {
        rand = new();
        prices = new() {
            { AAPL, 100m },
            { MSFT, 200m },
        };
    }
```

stock-ticker/Prices.cs (lines 21-36)

```
    internal decimal this[string symbol] {
        get {
            var delta = rand.Next(0, 100) switch {
                >= 95 => 0.2M,
                >= 80 => 0.1M,
                >= 70 => -0.1M,
                >= 65 => -0.2M,
                _ => 0M,
            };

            var newPrice = prices[symbol] + delta;
            prices[symbol] = newPrice > 0 ? newPrice : 0M;
            return prices[symbol];
        }
    }
}
```

For stock records, we define a new type `Stock` using the C# `record class`.

stock-ticker/Stock.cs (lines 3-12)

```
public record Stock(string Symbol, decimal Price, decimal PreviousPrice = 0M) {
    public string Trend => (Price - PreviousPrice) switch {
        > 0 => "Up",
        < 0 => "Dn",
        _ => "--",    // No change
    };

    public Stock UpdatePrice(decimal newPrice) =>
        this with { Price = newPrice, PreviousPrice = Price };
}
```

Clearly, it is not a coincidence that C# has chosen the word `record` for its new construct that can be used to create "data types". The word "record" in computing, as well as in the common English, has the connotations of "data".

18.3. Pair Programming - Primary Constructors, `switch` Expressions, `async` Methods

The `record`, or `record class`, provides a new way to create a reference type in C#. In fact, when a type is created using the `record (class)` declaration, the compiler creates a reference type using `class` behind the scene.

The compiler also generates a few synthesized methods to make a record type follow value equality semantics.

Records are typically used for "data" types in the same way that `readonly structs` are used. Structs are generally useful for "small" value types because copying large values can be expensive.

On the other hand, records are reference types. They follow mostly reference semantics, but the data (the references point to) need not be copied making their use more efficient for "large" data types.

Records follow value equality semantics. In many situations, they behave more like values than references.

Like structs, records are primarily used to create immutable types. As of C# 10.0, one can also create a `record struct`, which is a value type (`struct`) and which is syntactically similar to `record class`. The compiler generates the same set of methods for the `record structs` as well.

A `record` type can be created as follows:

```
record Stock {
    // ...
}
```

Or,

```
record class Stock {
    // ...
}
```

These two syntaxes are equivalent.

A record can include readonly or read-write properties as well as static and instance methods, etc. just like class or struct types.

Records (both record classes and record structs) support a special type of syntax, in which one or more readonly properties can be included in the parentheses after the type name.

A "readonly property" is a property with only a getter, In some contexts, an init-only property may also be considered readonly. In some other contexts, a property with a private setter may be considered readonly as well.

For instance,

```
record Stock(/* readonly properties here */) {
    // ....
}
```

This essentially defines a constructor for the record type.

In cases when a record type does not include any other methods or other properties, the whole block can be omitted.

```
record Stock(/* ... */);
```

A `record` defined in this syntax is called the "positional record". Note the semicolon at the end of the statement instead of the usual curly brace block.

In the sample code, `Stock` could have been declared as follows if it included no other properties or methods.

```
record Stock(string Symbol, decimal Price, decimal PreviousPrice =
0M);
```

Note the syntax. The readonly properties is listed in a pair of parentheses as if they are constructor arguments. One can set the initial/default values of the properties using a similar syntax as the optional arguments for the constructors.

The positional record syntax does define a constructor for the type. It is called the " primary constructor" of the record. Additional constructors can be defined.

Records can be created using the object initializer syntax as well just like `struct` and `class`.

The `Stock` record class defines four properties, `Symbol`, `Price`, `PreviousPrice`, and `Trend`.

Three of them are declared using the positional, or constructor, syntax. The `Trend` readonly property is defined using the more traditional `get` property syntax:

```
public string Trend {
    get => (Price - PreviousPrice) switch {
        > 0 => "Up",
        < 0 => "Dn",
        _ => "--",    // No change
    };
}
```

Since it is a readonly-property (that is, it has no setter), we use the shorthand notation in the sample code, using the expression body.

Note the use of the `switch` expression in the implementation of the` Trend` getter. It is as simple as using a ternary expression, and it is not limited to two alternative cases.

Since the `Trend` property returns a computed value, it may not have made sense to include it in the positional/constructor arguments. It is a readonly property (with no `init`) and it cannot be included in an object initializer.

Note that the (non-optional) positional arguments need to be provided via the constructor syntax.

For example, the following, using the named parameters, are all valid:

```
var stock = new Stock(Symbol: "AMZN", Price: 1.0M);
```

```
var stock = new Stock(Symbol: "AMZN", Price: 1.0M, PreviousPrice:
1.0M);
```

```
var stock = new Stock(Symbol: "AMZN", Price: 1.0M) {
    PreviousPrice = 1.0M,
};
```

If one prefers the object initializer syntax, then they can make all or most of the primary constructor parameters optional. Or, they can provide additional constructors that require no or fewer parameters.

The `Stock` record includes an instance method, `UpdatePrice()`:

```
public Stock UpdatePrice(decimal newPrice) => this with { Price =
newPrice, PreviousPrice = Price };
```

The implementation uses what is called the "non-destructive mutations". The data of the `Stock` object does not change when we call `UpdatePrice()`. Instead, it returns a *new* `Stock` object.

The `with` keyword is used to create a copy of an existing object (`this` object in this case) with a few select property values modified (inside the curly braces).

One can imagine that this can be very useful when dealing with large immutable record types. Instead of using a constructor or an object initializer, which may require initializing many variables, we simply use an existing record as a "template".

Furthermore, this syntax clearly indicates the underlying intention: The new object, or record, created this way is "based on" this particular existing object/record. Any properties that are not explicitly overwritten through this non-destructive mutation syntax will have the same values as those of the "template" object.

We have seen the use of the `this` keyword before, in the context of defining extension methods. That is a bit of a special case.

The keyword `this`, in general, is used to refer to "the current object", or "me", so to speak, inside a class, record, or struct definition. In particular, the `this` object is available in all instance properties and instance methods.

We have been carefully using the words, "types" and "objects", throughout this book. Unfortunately, the name of the base class in C# is `object`. This `object` is a type.

But, generally, we use the term "object" for an instance of a type. In fact, any type in C# is a child of the `object` type, and hence an instance of any type in C# is an "object".

More particularly, we often use the term "object" for an instance of a reference type. We do not generally refer to an `int` literal like `3` as an object. But, regardless, conceptually every value or reference in C# is an "object".

A "type" is a template, or blueprint, so to speak, for creating an object. That is, every object of the same type has been created from the same blueprint.

In the definition of a type (e.g., class, struct, or interface), the keyword `this` refers to the object that is created by this type, or the "object blueprint".

For example, in the definition of the sealed class `Prices`, we declare a couple of instance fields, namely, `rand` and `prices`.

(As stated a number of times before, anything "static" has little to do with the "object". Static fields, static properties, and static methods are not part of the "object blueprint", at least as of C# 10.0.)

One can refer to these instance fields, or just fields for short, using the "dot notation", in which we use the `this` keyword to refer to the (nameless) object.

For instance, the constructor can be written as follows:

```
private Prices() {
    this.rand = new();
    this.prices = new() {{ AAPL, 100m }, { MSFT, 200m }};
```

```
  }
```

The `this` keyword can be often omitted because in many cases it is obvious (to us as well as to the compiler) what object we are referring to. That is, `this.rand` refers to the `rand` field variable of "this object". In cases where it is not so obvious, for example when we have another variable with the same name `rand` within the same scope, we will need to include `this` with the dot notation.

You can refer to the fields, properties, and methods of an object, within the type definition, with the keyword `this`.

One thing to note about this particular constructor is that it is declared as `private`, which means that nobody can instantiate an object of this type. That is, nobody else but itself.

In fact, the `Prices` class creates one object, `instance`, and it exposes the object via the internal readonly property `Stock`. (Lines 7 and 8.)

This kind of type, or the only object of the type, is often referred to as a "singleton". One can have only one (or, a finite few) instance/object of a singleton type. It is common to have `private` constructors, or `protected` constructors, for the types like this which are not intended to be used to create many objects.

For a singleton object, it is customary to use a generic name like "Instance". In this particular example, the only use case of the `Prices` instance is to get the stock price through the "indexer syntax", and hence the name `Stock` seems to be more appropriate.

Speaking of the `indexer`, the `Prices` class shows another use of the `this` keyword.

In lines 21 through 35, it defines a readonly indexer. An `indexer` is a special kind of property.

If a type defines an indexer, then an object of that type can be used with the "index notation" as if it is a collection, e.g., like an array object.

Note the syntax for defining an indexer. It uses `this[…]` instead of the property name. Inside the square brackets, one can specify the "index" type and the variable name. Generally, `int`-based indexes are most common. But, C# allows different types like `string` as well.

In this particular example, the indexer is "readonly". There is only `get {}`, but not `set {}`.

```
decimal this[string symbol] {
    get {
        var newPrice = prices[symbol] + rand.Next(0, 100) switch {
            // ...
            _ => 0M,
        };
        prices[symbol] = newPrice > 0 ? newPrice : 0M;
        return prices[symbol];
    }
}
```

The details of this particular implementation are not important, but it adds a "random fluctuation" to the current price, using the switch expression, to get a new price. About 65% of the time, however, the price does not change (the default arm `_ => 0m`), in this particular implementation.

One thing to note here is that we use the field `prices` of type `Dictionary<string, decimal>` to "cache" the current price of each stock symbol.

There are many different types of "caching", used for a variety of purposes. This topic is, however, beyond the scope of this book. The interested readers are encouraged to learn more about the uses of caching, in general programming, from other resources.

In the `Ticker` static class, we get the price of a stock using the index notation, `Prices.Stock[Prices.AAPL]`. This is possible because the type `Prices` has the proper indexer property defined.

Note that the `Prices.Stock` is the property of an object of a type `Prices`, and its own type is also `Prices`, as we just discussed.

> This kind of construct is possible because `Prices` is a reference type. For value types, a type cannot include a variable of its own type. This is somewhat an advanced concept, and we will leave it to the readers, if interested, as to why that is, or why that *should be*, the case.
>
> The purpose of this book is not to teach *everything*, but just enough to help the readers get started in programming in C# with a solid foundation.

It should also be noted that the implementation of the `Ticker` class also does a kind of "caching" using the static field variable `stocks`. Again, we will leave it to the readers as to why this caching is "necessary" in this particular implementation.

When we run the main program, we get an output like this:

```
$ dotnet run

Stock { Symbol = AAPL, Price = 100.2, PreviousPrice = 100.2, Trend =
-- }
Stock { Symbol = MSFT, Price = 199.9, PreviousPrice = 200, Trend = Dn
}
Stock { Symbol = AAPL, Price = 100.4, PreviousPrice = 100.2, Trend =
Up }
Stock { Symbol = MSFT, Price = 199.9, PreviousPrice = 199.9, Trend =
-- }
```

```
Stock { Symbol = AAPL, Price = 100.5, PreviousPrice = 100.4, Trend =
Up }
Stock { Symbol = AAPL, Price = 100.6, PreviousPrice = 100.5, Trend =
Up }
Stock { Symbol = MSFT, Price = 200.1, PreviousPrice = 199.9, Trend =
Up }
Stock { Symbol = AAPL, Price = 100.7, PreviousPrice = 100.6, Trend =
Up }
Stock { Symbol = MSFT, Price = 200.1, PreviousPrice = 200.1, Trend =
-- }
Stock { Symbol = AAPL, Price = 100.7, PreviousPrice = 100.7, Trend =
-- }
Stock { Symbol = AAPL, Price = 100.6, PreviousPrice = 100.7, Trend =
Dn }
Stock { Symbol = AAPL, Price = 100.5, PreviousPrice = 100.6, Trend =
Dn }
Stock { Symbol = MSFT, Price = 200.0, PreviousPrice = 200.1, Trend =
Dn }
Stock { Symbol = AAPL, Price = 100.6, PreviousPrice = 100.5, Trend =
Up }
Stock { Symbol = MSFT, Price = 200.0, PreviousPrice = 200.0, Trend =
-- }
^C
```

We stopped the program using "Control-C" in this example.

A (fake) Apple `Stock` record is created using the `new` operator syntax, `new(Prices.AAPL, 0.0m)`.

Inside the `foreach` loop, we call the `Ticker.Get(symbol)` static method to get the current `Stock` record for the given `symbol`. We store them in a cache dictionary `stocks`, and print it out on console.

Note the `if` statement. This is done only if the current `Stock` record is different from the previously cached value for the given symbol.

```
if (stock != stocks[symbol]) {
    // ...
}
```

Although we have not explicitly defined the inequality operator (`!=`), we can use it for the `Stock` objects. This is because the compiler automatically generates equality and inequality operators for the record types (`record classes` and `record structs`). In fact, the compiler automatically synthesizes a number of methods for record types. Refer to the official C# reference for more information.

These synthesized method implementations ensure that the `record class` objects follow value equality semantics (even though they are intrinsically reference variables).

That is to say, in this particular example, the expression `stock != stocks[symbol]` will evaluate to `false` if the values of each of the properties of the `Stock` record type are all the same for the `stock` and `stocks[symbol]` variables.

This compiler help can greatly simplify the task of creating value or value-like types, or, "data types".

Summary

We used `record class` to create a custom type in this lesson. Records are useful in defining immutable data types which follow value equality semantics.

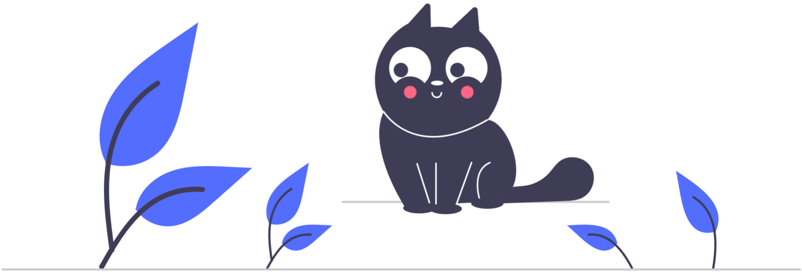
Chapter 19. Roman Numerals

19.1. Introduction

In the previous three lessons, we learned how to create custom types using `struct` and `class`, and their `record` cousins.

As stated, these are two of the most important building blocks in the C# programming language. (Or, four, depending on how we count them.) We will further explore these concepts in this lesson by creating a new type, `Roman.Numeral`, using each of these constructs.

Roman numerals is a very interesting way to write numbers. Integers between 1 and 3999 can be represented using Roman numerals. For more information, refer to this article on Wikipedia, for instance, Roman Numerals [https://en.wikipedia.org/wiki/Roman_numerals].



19.2. Code Review - Roman Numeral Converter

As stated, a C# program (or, a C# project) can have only one entry point. It can be the top-level statements in a single source file, or it can be the `Main()` static method of a class/record, struct/record struct, or an interface.

In this lesson, we basically create four distinct types to demonstrate the use of `struct`, `class`, and `record`. In the sample program, we define four different `Main()` methods for the four different types.

In order to be able to run the program, however, we will need to specify the main class. For this, we can use the `StartupObject` property in the C# project file. For example,

```
<StartupObject>Roman.CNumeral</StartupObject>
```

Alternatively, we can use the build option of `dotnet build` to specify the startup class. For example,

```
dotnet build -t:Rebuild -p:StartupObject=Roman.CNumeral
dotnet run
```

Note that when we change the startup class (or, struct, record, interface), we need to rebuild the program, e.g., using the `-t:Rebuild` flag.

> If you are using Visual Studio or other IDEs, then you can achieve more or less the same thing through the project settings or other options.

Here's a definition of `SNumeral` (*S* for `struct`). The `SNumeral` type is defined as a `struct`.

roman-numerals/SNumeral.cs (lines 3-24)

```
public readonly struct SNumeral {
    public static readonly SNumeral One = new(1);

    private readonly ushort number;

    public SNumeral(long number) => this.number = number switch {
        >= Constants.MIN and <= Constants.MAX => (ushort)number,
        _ => throw new ArgumentOutOfRangeException($"Roman numerals are between {Constants.MIN} and {Constants.MAX}"),
    };

    public SNumeral(string repr) =>
        number = NumeralHelper.Validate(repr) ? NumeralHelper.FromNumeralString(repr) :
            throw new ArgumentException($"Invalid Roman numeral: {repr}");

    public override string ToString() => NumeralHelper.ToNumeralString(number);

    public override int GetHashCode() => number.GetHashCode();

    public override bool Equals(object? obj) => obj is SNumeral snum && number == snum.number;

    public static bool operator ==(SNumeral l, SNumeral r) => l.Equals(r);
    public static bool operator !=(SNumeral l, SNumeral r) => !l.Equals(r);
```

roman-numerals/SNumeral.cs (lines 26-47)

```
    internal static void Main() {
        var r1 = new SNumeral(1234);
        Console.WriteLine($"r1 = {r1}");

        var r2 = new SNumeral(2022);
        Console.WriteLine($"r2 = {r2}");

        var r3 = new SNumeral("MMXXII");
        Console.WriteLine($"r3 = {r3}\n");

        Console.WriteLine($"{(r1 == r2 ? "r1 == r2" : "r1 != r2")}");
        Console.WriteLine($"{(r2 == r3 ? "r2 == r3" : "r2 != r3")}");
    }
}
```

The following is a definition of `CNumeral` (*C* for `class`). The `CNumeral` type is defined as a `class`.

roman-numerals/CNumeral.cs (lines 3-15)

```
public sealed class CNumeral {
    private readonly ushort number;

    public CNumeral(long number = 1) => this.number = number switch {
        >= Constants.MIN and <= Constants.MAX => (ushort)number,
        _ => throw new ArgumentOutOfRangeException($"Roman numerals are between {Constants.MIN} and {Constants.MAX}"),
    };

    public CNumeral(string repr) : this(
        NumeralHelper.Validate(repr) ? NumeralHelper.FromNumeralString(repr) :
            throw new ArgumentException($"Invalid Roman numeral: {repr}")) { }

    public override string ToString() =>
```

```
NumeralHelper.ToNumeralString(number);
```

Here are two other definitions of Roman number, `RcNumeral` (*rc* for `record class`) and `RsNumeral` (*rs* for `record struct`). The `RcNumeral` and `RsNumeral` types are defined as `class` and `struct`, respectively, with the `record` keyword.

roman-numerals/RcNumeral.cs (lines 3-18)

```
public sealed record RcNumeral() {
    private readonly ushort number;

    public RcNumeral(string repr) : this() =>
        number = NumeralHelper.Validate(repr) ? NumeralHelper.FromNumeralString(repr) :
            throw new ArgumentException($"Invalid Roman numeral: {repr}");

    public ushort Number {
        get => number;
        init => number = value switch {
            >= Constants.MIN and <= Constants.MAX => value,
            _ => throw new ArgumentOutOfRangeException($"Roman numerals are between {Constants.MIN} and {Constants.MAX}"),
        };
    }

    public string Roman => NumeralHelper.ToNumeralString(number);
```

roman-numerals/RsNumeral.cs (lines 3-18)

```
public readonly record struct RsNumeral {
    private readonly ushort number;

    public RsNumeral(string repr) : this() =>
        number = NumeralHelper.Validate(repr) ? NumeralHelper.FromNumeralString(repr) :
            throw new ArgumentException($"Invalid Roman numeral: {repr}");

    public ushort Number {
        get => number;
        init => number = value switch {
            >= Constants.MIN and <= Constants.MAX => value,
            _ => throw new ArgumentOutOfRangeException($"Roman numerals are between {Constants.MIN} and {Constants.MAX}"),
        };
    }

    public string Roman => NumeralHelper.ToNumeralString(number);
```

The actual implementations of converting from an integer to a Roman numeral string, and vice versa, are included in this helper class:

roman-numerals/NumeralHelper.cs (lines 5-9)

```
5 static class NumeralHelper {
6     private static Regex RnRx = new (@"^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$",
7         RegexOptions.Compiled | RegexOptions.IgnoreCase);
8
9     internal static bool Validate(string repr) => RnRx.IsMatch(repr);
```

The `Validate()` helper method that does the roman numeral validation uses the "regular expressions". We will not discuss the regular expressions in this book. They are supported pretty much across all different programming languages, and their uses are more or less the same.

If you are not familiar with regular expressions, then you can ignore this part. Or, you can look up the .NET reference manual on the Web.

roman-numerals/NumeralHelper.cs (lines 11-30)

```
    internal static ushort FromNumeralString(string repr) {
        var number = 0;

        char p = '\0';
        foreach (var c in repr) {
            number += c switch {
                'M' or 'm' => (p == 'C' || p == 'c') ? (1000 - 2 * 100) : 1000,
                'D' or 'd' => (p == 'C' || p == 'c') ? (500 - 2 * 100) : 500,
                'C' or 'c' => (p == 'X' || p == 'x') ? (100 - 2 * 10) : 100,
                'L' or 'l' => (p == 'X' || p == 'x') ? (50 - 2 * 10) : 50,
                'X' or 'x' => (p == 'I' || p == 'i') ? (10 - 2 * 1) : 10,
                'V' or 'v' => (p == 'I' || p == 'i') ? (5 - 2 * 1) : 5,
                'I' or 'i' => 1,
                _ => 0
            };
            p = c;
        }

        return (ushort)number;
    }
```

roman-numerals/NumeralHelper.cs (lines 32-86)

```
    internal static string ToNumeralString(ushort number) {
        var sb = new StringBuilder();

        while (number > 0) {
            if (number >= 1000) {
                sb.Append(number switch {
                    >= 3000 => "MMM",
                    >= 2000 => "MM",
                    _ => "M",
                });
                number %= 1000;
            } else if (number >= 100) {
                sb.Append(number switch {
                    >= 900 => "CM",
                    >= 800 => "DCCC",
                    >= 700 => "DCC",
                    >= 600 => "DC",
                    >= 500 => "D",
                    >= 400 => "CD",
                    >= 300 => "CCC",
                    >= 200 => "CC",
                    _ => "C",
                });
                number %= 100;
            } else if (number >= 10) {
                sb.Append(number switch {
                    >= 90 => "XC",
                    >= 80 => "LXXX",
                    >= 70 => "LXX",
                    >= 60 => "LX",
                    >= 50 => "L",
                    >= 40 => "XL",
                    >= 30 => "XXX",
                    >= 20 => "XX",
                    _ => "X",
                });
                number %= 10;
            } else {
                sb.Append(number switch {
                    >= 9 => "IX",
                    >= 8 => "VIII",
                    >= 7 => "VII",
                    >= 6 => "VI",
                    >= 5 => "V",
                    >= 4 => "IV",
                    >= 3 => "III",
                    >= 2 => "II",
                    _ => "I",
                });
                number = 0;
            }
        }

        return sb.ToString();
    }
```

These two static methods, `FromNumeralString()` and `ToNumeralString()`, implement conversions from a number (`ushort`) to its Roman numeral string representation, and vice versa. They can be implemented in a number of different ways.

In this example, we use the most straightforward implementations. The readers are encouraged to go through these two methods to see if they make sense.

The idea is the same as, or very similar to, that from one of the earlier lessons, in which we converted numbers between two different bases, e.g., 10 and 32 (Base 32 Numbers). The Roman numerals have a rather unique system of representing numbers unlike the number systems that we currently use.

The constants that are common to all four types are included in this small static class, for convenience. Normally, these constants would have been a part of a type definition, but in this example they are shared across four distinct types (which are essentially identical to one another other than the implementation details).

roman-numerals/Constants.cs (lines 3-6)

```
static class Constants {
    internal const ushort MIN = 1;
    internal const ushort MAX = 3999;
}
```

19.3. Pair Programming - Data Types, `switch` Expressions, Init-Only Properties

You can define an object type using any of `class`, `stuct`, `record`/`record class`, and `record struct` in C#.

As stated, `struct`, and `record struct`, creates a value type, which follows value semantics. In particular, the values of the struct types are "copied", and their equalities are to be evaluated based on their values, and their values alone.

A reference object type can be created using either `class` or `record`. The objects of the reference types follow reference semantics. That is, even when references are copied, they still point to the same underlying data.

As indicated in a previous lesson, the value types need to follow "value equality semantics". For a value object, the value *is* its "identity", and whether two value objects are the same or not (should) solely depend on their values, and nothing else. That is, `3` is `3`.

For all values types in C#, built-in or user-defined, the value equality is imposed, by default, by the implementation of `System.ValueType.Equals()`.

For reference types in C#, that is not the case. By default, by the implementation of `System.Object.Equals()`, all reference type objects merely follow reference equality semantics.

Note that the equality semantics mostly matters for data type objects. In C#, a `record` is a special kind of `class` that follows the value equality semantics by default.

We demonstrate the behaviors of `Equals()` or `==` with three different types based on `class`, `struct`, `record`, and `record struct`.

In the `Main()` method of `SNumeral`, we create three values, `r1`, `r2`, and `r3`, and compare their values.

If you run the program, as follows,

```
dotnet build -t:Rebuild -p:StartupObject=Roman.SNumeral
dotnet run
```

Here's a sample output.

```
$ dotnet run

r1 = MCCXXXIV
r2 = MMXXII
r3 = MMXXII

r1 != r2
r2 == r3
```

`r1` is not the same as `r2`, and `r2` is the same as `r3`, as expected.

Although `r2` and `r3` are created differently, and they are distinct variables, their values are the same, and they are equal to each other.

In the `SNumeral` struct, we do override `Equals()`, `operator ==()`, and `operator !=()`. But, that is just for convenience so that we can use the `r2 == r3` syntax. If we do not override these methods and just use `r2.Equals(r3)`, then we will still get the same result.

For value types, it is a good practice to override these three methods and/or implement the `IEquatable<T>` interface.

In the `Main()` method of `CNumeral`, we also create three values, `r1`, `r2`, and `r3`, in the exactly the same way, and compare their values.

If you run the program, as follows,

```
dotnet build -t:Rebuild -p:StartupObject=Roman.CNumeral
dotnet run
```

We get the following output:

```
$ dotnet run

r1 = MCCXXXIV
r2 = MMXXII
r3 = MMXXII

r1 != r2
r2 != r3
```

Even though `r2` and `r3` have the same value (or, they include the same data), the equality test returns `false`. It is not the value-equality semantics.

As indicated, this is not necessarily a problem. In many cases, it matters little. For "data" types, however, the value equality semantics is desirable.

C# now has a special kind of `class` that is a reference type, but supports the value equality semantics out of the box. We will look at a `record` type, `RcNumeral`, next.

In the `Main()` method of `RcNumeral`, we try the same things.

This time, we create three objects, `r1`, `r2`, and `r3`, using the constructor and the object initializer syntax, just for a change, and compare their values.

If you run the program as before:

```
dotnet build -t:Rebuild -p:StartupObject=Roman.RcNumeral
dotnet run
```

Here’s the sample output:

```
$ dotnet run

r1 = RcNumeral { Number = 1234, Roman = MCCXXXIV }
r2 = RcNumeral { Number = 2022, Roman = MMXXII }
r3 = RcNumeral { Number = 2022, Roman = MMXXII }

r1 != r2
r2 == r3
```

Although these are reference variables, they follow the value-equality semantics. In particular, `r2` and `r3`, which are distinct reference variables which point to two different underlying data (e.g., in two different memory locations), are *considered equal* in this implementation using records, by default.

Note that we get a reasonable implementation of the `ToString()` method for free (based on the two public properties that we declare, `Number` and `Roman`) when we use records (unlike in the case of `structs` or `classes`).

We can still override this method if a different output format is desired. For example, as before,

```
public override string ToString() =>
NumeralHelper.ToNumeralString(number);
```

Note the interesting syntax when we define the `RcNumeral` record:

```
public record RcNumeral() {
    // ..
}
```

This uses the positional record syntax *without any property arguments.* This declaration still gives us a constructor (with no parameters) as the *primary constructor*.

In the example `Main()` method, we use the object initializer syntax, e.g.,

```
var r1 = new RcNumeral { Number = 1234 };
```

We could not have used this syntax without the default constructor explicitly defined (through the positional argument syntax).

For `struct` types, the default (no-argument) constructors are always created by the compiler, and they cannot be overridden.

For `class`, and `record`, types, the default constructors are created when no other constructors are explicitly defined in the type definition. If there is any constructor, then the default constructor is not generated.

In the case of `record structs`, they are value types just like `structs`, and they follow the same value semantics. They provide some convenience, over the plain `structs`, in terms of syntax and what not.

For completeness, if we build and run our example program with `RsNumeral` as the main class,

```
dotnet build -t:Rebuild -p:StartupObject=Roman.RsNumeral
dotnet run
```

We get the following output, which is the same as those from `SNumeral` and `RcNumeral`.

```
r1 = RsNumeral { Number = 1234, Roman = MCCXXXIV }
r2 = RsNumeral { Number = 2022, Roman = MMXXII }
r3 = RsNumeral { Number = 2022, Roman = MMXXII }

r1 != r2
r2 == r3
```

We have tried implementing the "Roman numeral" type in this lesson using four different constructs, `class`, `struct`, `record`, and `record struct`. In this particular example, the use of the `struct`, or `record struct`, seems to be most reasonable. It is a "small" type, and it is rather natural to use value semantics for number types.

Summary

We did simple comparisons among `class`, `struct`, `record class`, and `record struct` by implementing a type for Roman numerals. Types created with either `struct`/`record struct` or `record` follow value equality semantics and they are most suitable for "data types".

Chapter 20. Euclidean Distance

20.1. Introduction

C#'s new `record class` seems to provide the best of both worlds, value types and reference types. (Or, does it?)

We will again try using both `struct` and `record` (which is just `class` with special compiler support) for implementing data types.

The rule of thumb is to use `structs`, or `record structs`, for "pure" value or "small" data types and `records` for "large" data types.

> We often use the imprecise terms like "small" and "large" throughout this book. Unfortunately, the readers will have to develop certain "intuitions" through experience. These concepts are context-dependent, and they cannot be really precisely defined.

In this lesson, we will implement a small type "point" to represent a point in the two dimensional space. A point can be viewed as a pure value type, like a complex number, or as a data type, like a geographical coordinate, depending on the circumstances.

We also introduce the important concept of `interface` in this lesson. We will use a simple interface to provide a polymorphic behavior to the point types that we create using `struct` and `record (class)`.

> The example that we use in this lesson seems to be a bit "contrived". In practice, we will rarely define more than one types for the exactly same underlying concept in the same program. Nonetheless, this code sample illustrates an important use of

> interface in addition to demonstrating some essential uses of struct vs record.

20.2. Code Review

A "point" is a fundamental concept. Just like a number.

We live in a three-dimensional space, and we often deal with (conceptually) two-dimensional surfaces like a notepad, a map, a computer screen, etc. Let’s try representing a point in a 2-D space as a pair of numbers.

Traditionally, we would have most likely used struct to define a type for "points" in C#. Here’s a definition of SPoint.

euclid-distance/SPoint.cs (lines 3-41)

```
public readonly struct SPoint : IPoint, IEquatable<SPoint>,
IComparable<SPoint> {
    public float X { get; init; }
    public float Y { get; init; }

    public SPoint(float x, float y) => (X, Y) = (x, y);

    public float Radius() => (float)Math.Sqrt(X * X + Y * Y);

    public override string ToString() => $"({X},{Y})";

    public override bool Equals(object? obj) =>
        obj is SPoint point && X == point.X && Y == point.Y;

    public override int GetHashCode() => HashCode.Combine(X, Y);

    public bool Equals(SPoint other) => X == other.X && Y == other.Y;

    public static bool operator ==(SPoint left, SPoint right) => left.Equals(right);

    public static bool operator !=(SPoint left, SPoint right) => !(left == right);

    public int CompareTo(IPoint? other) => other is SPoint sp ? CompareTo(sp) : 1;

    public int CompareTo(SPoint other) => (Radius() - other.Radius()) switch {
        < 0 => -1,
        > 0 => 1,
        _ => 0,
    };

    public static bool operator >(SPoint left, SPoint right) => left.Radius() > right.Radius();
    public static bool operator <(SPoint left, SPoint right) => left.Radius() < right.Radius();
    public static bool operator >=(SPoint left, SPoint right) => left.Radius() >= right.Radius();
    public static bool operator <=(SPoint left, SPoint right) => left.Radius() <= right.Radius();

    public IPoint Copy() => Copy((0, 0));
    public IPoint Copy((float, float) delta) => new SPoint(X + delta.Item1, Y + delta.Item2);

    public void Deconstruct(out float X, out float Y) => (X, Y) = (this.X, this.Y);
}
```

Here's an alternative type definition, `RPoint`, using `record class` for representing points in a two dimensional space.

euclid-distance/RPoint.cs (lines 3-21)

```
public record RPoint(float X, float Y) : IPoint, IComparable<RPoint> {
    public float Radius() => (float)Math.Sqrt(X * X + Y * Y);

    public int CompareTo(IPoint? other) => other is RPoint rp ? CompareTo(rp) : 1;

    public int CompareTo(RPoint? other) => other is null ? 1 : (Radius() - other.Radius()) switch {
        < 0 => -1,
        > 0 => 1,
        _ => 0,
    };

    public static bool operator >(RPoint left, RPoint right) => left.Radius() > right.Radius();
    public static bool operator <(RPoint left, RPoint right) => left.Radius() < right.Radius();
    public static bool operator >=(RPoint left, RPoint right) => left.Radius() >= right.Radius();
    public static bool operator <=(RPoint left, RPoint right) => left.Radius() <= right.Radius();

    public IPoint Copy() => Copy((0, 0));
    public IPoint Copy((float, float) delta) => new RPoint(X + delta.Item1, Y + delta.Item2);
}
```

In fact, the (abstract) notion of a "point" can be encapsulated as follows, using `interface`. An `interface` in C# defines a set of "behaviors", but not "data".

euclid-distance/IPoint.cs (lines 3-13)

```
public interface IPoint : IComparable<IPoint> {
    float X { get; }
    float Y { get; }

    float Radius();

    IPoint Copy();
    IPoint Copy((float, float) delta);

    void Deconstruct(out float X, out float Y);
}
```

The prefix `I` (for `I`nterface) is conventional, and it is not required. This prefix is often used when we define both interface and concrete type, and the `I` prefix can be useful to distinguish the interfaces from other concrete types.

One thing to notice from the definition of `IPoint` is the property declarations. For example,

```
float X { get; }
```

This declares a (readonly) property X of type float in an interface. As you may recall from the earlier lessons, the exactly same syntax is used in the "auto property" declaration in a class or struct (or, their record counterparts). This can be a cause of confusion for some people who are new to C#.

This syntax in an interface does not define an auto property since the interfaces cannot have data/fields. It is an implicit "abstract property". Any (non-abstract) class that implements this interface will need to implement this property.

Using this new type IPoint, we can provide an implementation for more than one types, namely, for both SPoint and RPoint.

The following function, or static method, computes a Euclidean distance between two points, and it can be used for both SPoint and RPoint.

euclid-distance/Distance.cs (lines 4-5)

```
    public static float Distance(IPoint p1, IPoint p2) =>
        (float)Math.Sqrt((p1.X - p2.X) * (p1.X - p2.X) + (p1.Y -
p2.Y) * (p1.Y - p2.Y));
```

Here’s a method to sort an array of points, implemented generically for IPoint.

euclid-distance/Bubble.cs (lines 3-25)

```
static class Bubble {
    internal static void Sort<T>(IList<T> arr) where T : IPoint {
        for (var i = 0; i < arr.Count - 1; i++) {
            for (var j = 0; j < arr.Count - i - 1; j++) {
                if (Compare(arr[j], arr[j + 1]) > 0) {
                    (arr[j], arr[j + 1]) = (arr[j + 1], arr[j]);
                }
            }
        }
    }

    private static int Compare<T>(T l, T r) where T : IPoint {
        var (dl, dr) = (l.Radius(), r.Radius());
        return (dl - dr) switch {
            < 0 => -1,
            > 0 => 1,
            _ => 0,
        };
    }

    internal static void Print<T>(this IEnumerable<T> arr) =>
        Console.WriteLine($"{{{string.Join(", ", arr)}}}");
}
```

20.3. Pair Programming - Interfaces, Polymorphism, Custom Tuple Deconstruction

An `interface` is a reference type. The C# keyword `interface` is used to define a reference type.

Unlike the types defined by `struct` and `class`, an `interface` type is always an " abstract type". It defines a set of "behaviors", and it can provide some default implementations, but the objects of an interface type cannot be directly instantiated. Interfaces do not include "states" or "data". They only prescribe "behaviors".

Interfaces provide "polymorphic behavior" to other types. Types defined with `class`/`record` and `struct`/`record struct` can inherit behaviors from interfaces, e.g., by "implementing the interfaces".

C# also provides another way to define a type which is primarily behavior, namely `abstract class`. An abstract class defines a type that cannot be directly instantiated like `interface`. But, it can still include states unlike the `interface`. We will see example uses of `abstract class` in later lessons.

The `IPoint` public interface of the sample program defines a set of behaviors, i.e., properties and methods, that encapsulate the abstract concept of the "point" in the 2-D space.

That is, a point comprises two numbers (e.g., `float` in this example) and they are publicly accessible (e.g., through properties named `X` and `Y`).

```
float X { get; }
float Y { get; }
```

Given a point, one can obtain a certain number called the "radius" of the point.

```
float Radius();
```

In C#, properties and methods are more or less equivalent to each other despite their syntactic differences. The "methods" that (directly) access "data" are typically implemented via readonly or read-write properties. The `Radius()` method in this example could have been declared as a readonly property. In fact, that could have been more appropriate since the concept of the

> "radius" of a point is really "data".
>
> In many cases, it is just a matter of style or preference. One thing to note, however, is that the methods in an interface can have default implementations. On the other hand, the properties in an interface are always abstract.

Points can be "copied", either at the same location or at a new location displaced by "delta":

```
IPoint Copy();
IPoint Copy((float, float) delta);
```

And, they can be "deconstructed" into a tuple of two `float` numbers.

```
void Deconstruct(out float X, out float Y);
```

In fact, we could have even provided its default implementation for all types that inherit the behaviors of this interface. For instance,

```
void Deconstruct(out float X, out float Y) {
    (X, Y) = (this.X, this.Y);
}
```

Any type that implements this interface can use this default implementation without having to implement its own version. They can also "override" the default implementations if they need to.

In C#, interfaces can even include static method implementations (with various access modifiers). Static methods are, strictly speaking, not a part of a "type".

As mentioned, however, the "abstract static methods" (in interfaces), which will likely be introduced in C# 11.0, should be considered a part of a type since they define behaviors of the objects.

Note that all properties and (non-static) methods in an interface are `public` by default, and they do not need the `public` access modifier. They cannot be `private` unless they have implementations.

Comments, and documentations, are a vey important part of program source code, especially, for the public APIs. As stated, however, we do not include comments in the code samples in this book to focus on the teaching points and possibly to reduce the (unnecessary) clutter.

In addition, `IPoint` "extends" the interface `IComparable<IPoint>` (using the `:` syntax), which defines one method `int CompareTo (IPoint? other)`. We used this interface in earlier lessons.

The `IPoint` type, therefore, "inherits" this behavior from `IComparable<IPoint>`. As stated, sorting methods, for instance, require their element types to be " comparable".

An `interface` type can be used anywhere we can use a type, just like a delegate. In fact, a delegate can be conceptually viewed as a special kind of an interface (that defines a single method).

An object of a given interface type cannot be directly instantiated, however. (And, as a corollary, an interface cannot have constructors.) For that, we need to use a concrete type that "implements" the given interface.

For example, the type `SPoint` in this lesson implements the interface `IPoint`. Any object of a type `SPoint` is therefore an object of a type `IPoint`.

The `SPoint` type implements all properties and methods declared in `IPoint`, and more. In particular, it includes a public constructor:

```
SPoint(float x, float y) => (X, Y) = (x, y);
```

Note that this expression body syntax is equivalent to the following:

```
public Point(float x, float y) {
    (X, Y) = (x, y);
}
```

This is in turn equivalent to the following:

```
public Point(float x, float y) {
    X = x;
    Y = y;
}
```

As stated, there is no difference among these definitions as far as the compiler is concerned. It will generate the same IL code or an executable. It is really a matter of an individual's, or a team's, preference as to which style to prefer.

The `SPoint` type additionally implements the `IComparable<SPoint>` interface as well as the `IEquatable<SPoint>` interface (again using the `:` syntax), which includes one method `bool Equals (SPoint? other)`.

Any (value-semantics) type, or "data" type, that can be used in a context where equality comparisons may be required should implement `IEquatable<T>`. For example, objects of a type that does not support `IEquatable<T>` cannot be used as entries in the .NET generic collections such as `List<T>` or `Dictionary<K,V>`.

The type `RPoint` also implements `IPoint` as well as `IComparable<RPoint>`.

One of the nicest things about using `record` is that the compiler automatically generates a number of methods for us. In particular, a record type automatically provides implementations for `IEquatable<T>`, or `IEquatable<RPoint>` in this example.

It also syntheses a universal `ToString()` method based on the public properties of the record. One can always override it if necessary/desired. For instance,

```
public override string ToString() => $"({X},{Y})";
```

It also automatically includes the `Deconstruct()` method if we define a record type using the positional record syntax, as we do in this example:

```
record RPoint(float X, float Y);
```

As stated, a default implementation of `Deconstruct()` could have been provided in the definition of the `IPoint` interface, in our particular sample code.

Note that we do not declare this record `sealed` in this example. Unlike `struct`, records can be inherited. For instance, we can easily define a 3-D point type from this `RPoint` type, e.g., just by adding an additional property `float Z`, and overriding some implementations, etc.

For example,

```
record RPoint3D(float X, float Y, float Z) : RPoint(X, Y);
```

We will leave its full implementation as an exercise to the readers.

The `Distance()` static method is defined in terms of `IPoint` type. This is one of the "polymorphism" supported by the C# programming language.

For instance, when values of `SPoint` are passed in to the function, the function behaves just like it has been defined for two `SPoint` objects.

```
public static float Distance(SPoint p1, SPoint p2) =>
    (float)Math.Sqrt((p1.X - p2.X) * (p1.X - p2.X) + (p1.Y - p2.Y) *
(p1.Y - p2.Y));
```

Likewise, for `RPoint` objects, it behaves as if it has been defined for that type:

```
public static float Distance(RPoint p1, RPoint p2) =>
    (float)Math.Sqrt((p1.X - p2.X) * (p1.X - p2.X) + (p1.Y - p2.Y) *
(p1.Y - p2.Y));
```

One thing to note is that we could have used generics for this purpose as well. For example, over a generic type that "is" an `IPoint` (or implements `IPoint`).

```
public static float Distance<T>(T p1, T p2) where T : IPoint {
    // ...
}
```

Generics provides a different kind of polymorphic behavior, but it can be used to achieve the same things as using inheritance- or interface-based polymorphisms in some cases (but, not always).

Note that the generic methods/classes generate specializations at build time, whereas inheritance-based polymorphism is a runtime behavior.

As alluded before, C# provides an extremely powerful generics, unparalleled in any other modern programming languages except for maybe C++20, and programmers should take advantage of the generics more than anything else, if possible.

In fact, we implement a sorting method using generics in this lesson. The `Bubble.Sort<T>` static method is based on the version that we used in part 1. It is simplified. (There is no "pivot".) The sorting order, in this example, is based the "radius" (the distance of a point from the origin `(0,0)`).

Although it is a simple example, this method would have been more difficult to efficiently implement, and use, for both `SPoint` and `RPoint` using the interface-based polymorphism, e.g., using `IList<IPoints>`.

> ℹ We will not go into details of the advanced concepts like " variance", e.g., covariance and contravariance, in this book. Readers, if interested, are encouraged to explore further on this (very important) topic.

One other thing to note is that `SPoint` is a value type whereas `IPoint` is a reference type. By using `IPoint` for `SPoint`, we may end up losing some benefits of using value types.

C# does the so-called "boxing" and "unboxing" operations when the variables are converted between the value and reference types at *run time*.

The values stored on the stack memory, for instance, can be moved to the heap memory through "boxing". An "unboxing" refers to the reverse operation.

The implementation of `Bubble.Sort()` is more or less the same (only simpler) as the previous generic implementation for the "number" types. It uses the "bubble sort" algorithm.

Note the use of the switch expression:

```
return (dl - dr) switch {
    < 0 => -1,
    > 0 => 1,
    _ => 0,
};
```

This is simpler and easier-to-read, and "more elegant", than using the double ternary operator expression, as in our earlier example.

```
return dl.Equals(dr) ? 0 : (dl.CompareTo(dr) > 0 ? 1 : -1);
```

The `switch` expression can be viewed as a generalization of the ternary operator, and it is much more powerful.

One can also use `if`/`else` statements, if one prefers.

The "modern C#" provides choices, a lot of choices, and programmers should take advantage of that flexibility, rather than using C# just like any other C-style language.

The main program tests various behaviors of these point types:

euclid-distance/Program.cs (lines 4-12)

```
var p1 = new SPoint(10, -10);
var p2 = new SPoint(10, 20);
var p3 = new SPoint(0, 40);
Console.WriteLine($"p0 = {p0}; p1 = {p1}; p2 = {p2}; p3 = {p3}");

Console.WriteLine($"{(p0 == p2 ? "p0 == p2" : "p0 != p2")}");

TestPointType(p0, p1, p2, p3);
```

```
Console.WriteLine();
```

euclid-distance/Program.cs (lines 16-24)

```
var q1 = new RPoint(10, -10);
var q2 = new RPoint(10, 20);
var q3 = new RPoint(0, 40);
Console.WriteLine($"q0 = {q0}; q1 = {q1}; q2 = {q2}; q3 = {q3}");

Console.WriteLine($"{(q0 == q2 ? "q0 == q2" : "q0 != q2")}");

TestPointType(q0, q1, q2, q3);
```

euclid-distance/Program.cs (lines 27—51)

```
    var dist = Points.Distance(p[0], p[1]);
    Console.WriteLine($"dist(p0, p1) = {dist}");

    var radius = p[2].Radius();
    Console.WriteLine($"radius(p2) = {radius}");

    var (px0, py0) = p[0];
    Console.WriteLine($"px0 = {px0}; py0 = {py0}");

    Console.WriteLine($"{(p[0] == p[2] ? "ip0 == ip2" : "ip0 != ip2")}");

    var eql = (p[0], p[2]) switch {
        (SPoint sp0, SPoint sp2) => sp0 == sp2,
        (RPoint rp0, RPoint rp2) => rp0 == rp2,
        _ => p[0] == p[2],
    };
    Console.WriteLine($"{(eql ? "p0 == p2" : "p0 != p2")}");

    var copied = p[3].Copy((1.5f, 5));
    Console.WriteLine($"copied = {copied}");

    Bubble.Sort(p);
    p.Print();
}
```

If we run the program with `dotnet`,

```
dotnet run
```

We get the following output:

```
p0 = (10,20); p1 = (10,-10); p2 = (10,20); p3 = (0,40)
p0 == p2
dist(p0, p1) = 30
radius(p2) = 22.36068
px0 = 10; py0 = 20
ip0 != ip2
p0 == p2
copied = (1.5,45)
{(10,-10), (10,20), (10,20), (0,40)}

q0 = RPoint { X = 10, Y = 20 }; q1 = RPoint { X = 10, Y = -10 }; q2 =
RPoint { X = 10, Y = 20 }; q3 = RPoint { X = 0, Y = 40 }
q0 == q2
dist(p0, p1) = 30
radius(p2) = 22.36068
px0 = 10; py0 = 20
ip0 != ip2
p0 == p2
```

```
copied = RPoint { X = 1.5, Y = 45 }
{RPoint { X = 10, Y = -10 }, RPoint { X = 10, Y = 20 }, RPoint { X =
10, Y = 20 }, RPoint { X = 0, Y = 40 }}
```

Readers are encouraged to go though the main program and see what exactly is happening in this program. But, for one thing, note that both `SPoint` and `RPoint` follow value-equality semantics. That is, `p0 == p2`.

On the other hand, the reference type `IPoint` would not have displayed value-equality semantics.

Notice how we compare `p[0]` and `p[2]` in the `TestPointType()` method. A simple equality comparison would have resulted in `p0 != p2` because `p[0]` and `p[2]` are the variables of `IPoint`, a reference type.

The `TestPointType()` method also uses the C# keyword `params` to define a "vararg" function.

Summary

We explored some essential properties and uses of the interfaces in this lesson. One can provide a polymorphic behavior for different types (even for the value types defined with `struct` and `record struct`) using `interfaces`.

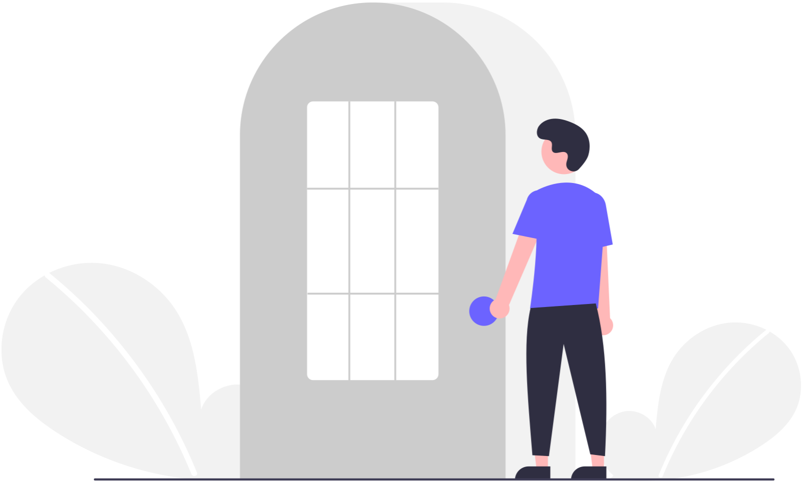
Chapter 21. Smart Door Lock

21.1. Introduction

Let’s suppose that we are creating a software to control a "smart door lock", which requires a numeric secret code to unlock the door.

For simplicity, we assume that a single code allows a user to unlock the door, or to reset the current secret code.

Users can lock the door without the code.



21.2. Code Review - Electronic Door Lock Controller

The requirements of this exemplary door lock are "behaviors". Doors open and close. Door locks lock and unlock.

A door lock can keep an internal state (e.g., the secret code), but that is not really "data".

We use many terms without precise definitions in this book, but the word "data" often means the data that can be, for example, transmitted between two entities, or can be stored as numbers or strings, etc.

A door lock is not "data". Roughly speaking. Regardless, a smart door lock is primarily about the "behavior" not the "data".

The appropriate construct to use to encapsulate an object like a smart door lock is therefore `class`. We *can* use `struct` or `record` to define a smart door lock type, but their value semantics or data semantics may not be very useful for door locks.

The rule of thumb is to just use `class` unless the abstract concept, a thing, that we are trying to encapsulate as a type is primarily a "value" or "data".

The main program tests a few APIs of our smart door lock, `Smart.DoorLock`, especially, the `Lock()` and `Unlock()` methods.

smart-door-lock/Program.cs (lines 3-17)

```
var code = "1234";
using var doorLock = new DoorLock(code);

var locked = doorLock.Lock();
Console.WriteLine($"locked =\t{locked}");

var unlocked = doorLock.Unlock(code);
Console.WriteLine($"unlocked =\t{unlocked}");

var newCode = "9876";
var codeSet = doorLock.SetCode(currentCode: code, newCode: newCode);
Console.WriteLine($"codeSet =\t{codeSet}");
```

```

var unlocked2 = doorLock.Unlock(code);
Console.WriteLine($"unlocked2 =\t{unlocked2}");
```

The `DoorLock` type in the `Smart` namespace exposes a few public methods as well as an auto property, `State`.

smart-door-lock/DoorLock.cs (lines 5-58)

```
public sealed class DoorLock : IDisposable {
    private readonly SHA256 crypto;
    private string hash;

    public LockState State { get; private set; }

    public DoorLock(string initialCode) {
        if (!IsCodeInValidFormat(initialCode)) {
            throw new ArgumentException($"Invalid code: {initialCode}");
        }
        crypto = SHA256.Create();
        hash = crypto.ComputeHash(Encoding.ASCII.GetBytes(initialCode)).ToHex();
        State = LockState.Locked;
    }

    public void Dispose() => crypto.Dispose();

    public bool Lock() {
        if (State == LockState.Locked) {
            return false;
        }
        State = LockState.Locked;
        return true;
    }

    public bool Unlock(string code) {
        if (State == LockState.Unlocked) {
            return false;
        }
        if (crypto.ComputeHash(Encoding.ASCII.GetBytes(code)).ToHex() == hash) {
            State = LockState.Unlocked;
            return true;
        }
        return false;
    }

    public bool SetCode(string newCode, string currentCode) {
        if (crypto.ComputeHash(Encoding.ASCII.GetBytes(currentCode)).ToHex() == hash) {
            if (!IsCodeInValidFormat(newCode)) {
                return false;
            }
            hash = crypto.ComputeHash(Encoding.ASCII.GetBytes(newCode)).ToHex();
            State = LockState.Locked;
            return true;
        }
        return false;
    }

    private static bool IsCodeInValidFormat(string code) {
        if (string.IsNullOrWhiteSpace(code) || code.Trim() != code) {
            return false;
        }
        return ulong.TryParse(code, out ulong _);
    }
```

Note that `DoorLock` inherits the behavior of `IDisposable`, which we used in some earlier lessons.

The door lock state, `LockState`, is defined as an `enum`:

smart-door-lock/State.cs (lines 3-6)

```
public enum LockState : ushort {
    Unlocked = 0,
    Locked,
}
```

As stated, an `enum` creates a value type, in particular, an integral type.

Here's a simple helper extension method. An extension method can only be defined in a static class.

smart-door-lock/Helper.cs (lines 5-11)

```
    internal static string ToHex(this byte[] bytes) {
        var sb = new StringBuilder();
        for (int i = 0; i < bytes.Length; i++) {
            sb.Append(bytes[i].ToString("X2"));
        }
        return sb.ToString();
    }
```

One cannot add additional behaviors to an enum type, e.g., by using interfaces or instance methods, beyond what is provided as

> an integral type. However, as indicated before, one can define extension methods on enum types.

21.3. Pair Programming - Interfaces, Extension Methods, Cryptography, `IDisposable`

If we run this program, then we get the following output:

```
$ dotnet run

locked =        False
unlocked =      True
codeSet =       True
unlocked2 =     False
```

The code sample mostly uses the features of C# that we already discussed.

The `DoorLock` type is defined as a public `sealed class`. It implements `IDisposable`, which declares one method `void Dispose()`.

In this example, the `Dispose()` method is used to clean up the crypto hashing object `SHA256`. This is similar to `MD5` that we used before. They essentially expose the same public APIs in .NET. `SHA256` uses "more bits" and it is more secure than `MD5`.

Syntactically, that is still a valid syntax. E.g.,

```
using var lock = new DoorLock(code) {};
```

Note the use of the `using` statement. As discussed, the `Dispose()` method of `DoorLock` would be automatically called when the variable `doorLock` goes out of the scope, at the end of the (implicitly defined) `Main()` method in this example.

> There is a concept of "finalizer" in C#. But, they are rarely useful in C# programs, unlike the "destructors" in C++.
>
> In many cases, if you need a resource cleanup (other than the heap memory), then use the `IDisposable` interface.

Note that since we have explicitly defined at least one constructor, `DoorLock(string initialCode)` in the `DoorLock` class, the default constructor will not be automatically generated by the compiler. In this example, this constructor that takes one string argument is the only constructor available.

Using an object initializer would not make sense in this case since there are no public properties that can be set (e.g. via `{ get; set; }` or `{ get; init; }`) via the object initializer syntax.

Note the use of `_` as an `out` parameter in the definition of the `IsCodeInValidFormat()` static method.

```
return ulong.TryParse(code, out ulong _);
```

We simply use the `System.UInt64.TryParse()` method to validate the numerical code, but we do not use the parsed value. We just discard them. Hence _ is called a discard variable.

It can also be used in situations like deconstruction when we are not interested in using all the deconstructed values.

> There is a very interesting use of discard variables. As stated, not

> all expressions in C# can be used as stand-alone statements. In fact, only assignment, method call, increment, decrement, await, and `new` object expressions can be used as statements.
>
> There is a workaround. You can use *any expression* on the right hand of an assignment and use the discard variable on the left hand side. Now you have a statement that is essentially an expression (since the assignment has no effect).
>
> (Not that we are recommending such uses in general, but there are situations where this trick comes in handy.)

The `bool SetCode(string newCode, string currentCode)` public method of `DoorLock` takes two arguments of the same type `string`. In this particular implementation, the "newCode" comes before the "currentCode".

This can be confusing when we use the `SetCode()` method. Which one comes first?

In situations like this, and when there are many (optional) arguments in a method, etc., it is sometimes best to use "named parameters".

Named parameters do not have to follow the positional order of the argument list as long as they are not followed by out-of-order positional parameters. Positional parameters can precede named parameters. Or they can even follow named parameters as long as those named parameters happen to be in the respective correct positions.

In the main method, we call `SetCode()` with named parameters. In this case, since we only use the named parameters, the order of these parameters does not matter:

```
var codeSet = doorLock.SetCode(currentCode: code, newCode: newCode);
```

One other thing to note in this sample code is that we could have thrown exceptions when the methods like `Unlock()` and `SetCode()` fail.

```
public bool Unlock(string code) {
    // if a valid code is not provided,
    throw new InvalidOperationException("Cannot unlock the door
without a valid code.");
}
```

```
public bool SetCode(string newCode, string currentCode) {
    // if a valid code is not provided,
    throw new UnauthorizedAccessException("Cannot set a new code
without the current code");
}
```

Then we could have used the `try/catch/finally` construct to check the "error" conditions. For example,

```
try {
    var unlocked2 = doorLock.Unlock(code);
    Console.WriteLine($"unlocked2 = {unlocked2}");
} catch (Exception ex) {
    Console.WriteLine(ex);
}
```

As a general rule of thumb, however, use of exceptions should be generally avoided unless it is truly an exceptional situation which the programmer cannot expect at the time of creating the software, etc.

Network errors, or file I/O errors, etc. cannot be really anticipated by their very nature. We just throw an exception and hope that the calling method has a better context to know what to do in those situations.

There are also performance implications. In earlier examples, for example, we used the variants of `TryParse()` methods rather than the `Parse()` methods that throw exceptions.

In this particular example, getting an invalid code from a user is a part of the "normal" operations. Although it is a matter of preference, to a large extent, we instead choose to return a boolean `false` value rather than throwing an exception in this example.

> Although we did not really discuss the `try/catch/finally` syntax in depth in this book. The C#'s try/catch/finally is more or less the same as those found in other (C-style) languages.
>
> We normally include the code that can potentially throw an exception in a `try` block. The try block can be followed by one or more `catch` blocks or a `finally` block, or both.
>
> The statements within the catch block is executed if the specified exception is thrown from the code in the try block. The statements in the finally block are always executed regardless of whether an exception is thrown, If an exception is caught in any of the catch blocks, the code of the finally block, if present, is executed after the code of that catch block is executed.
>
> C# provides a very powerful syntax for the exception handling.
>
> As alluded before, however, the importance of the "exceptions" in the general error handling in the C# programs has been gradually diminishing. For example, in certain cases, returning an error value or signal via the normal call chains is preferred over throwing an exception.

Summary

We reviewed the uses of `class` and `enum` in this lesson. We also looked at the `IDisposable` interface and the `using` statements.

Chapter 22. Orgs and Employees

22.1. Introduction

As stated, C# is an *object-oriented programming (OOP) language*. That is, C#, as a programming language, has many features that support the object-oriented programming style.

First and foremost, it supports "data encapsulation" (or, "data hiding").

The modern C# prefers properties, or auto-properties, over the use of explicit data fields. Although, in theory, one can still have public fields, such practices are all but disappeared from the C# universe. Data is always accessed via a well-defined, and access-controlled, interface such as properties or methods.

We have seen some of such examples in this book. The three main constructs of C#, `class`, `record`, and `struct`, all support data encapsulation.

C# also supports inheritance-based polymorphism (aka "subtyping") via interfaces and other reference types like classes and records, as we indicated in the earlier lessons.

> Note that this kind of polymorphism only works with reference type variables. C#'s value objects, defined by structs or enums, do not, and cannot, support such polymorphic behavior.
>
> This is not the limitation of C#. This is true for all OOP languages. For example, in C++ and Rust, if you want to use polymorphism, you will need to use pointer/reference types.
>
> This is because, roughly speaking, the polymorphism relies on some kind of "indirection". A reference variable points to a value

> in other location. A value variable does not.

In this lesson, we will briefly touch on a number of features in C# that support the inheritance-based polymorphism. We already discussed its closely related cousin, the interface-based polymorphism in the earlier lessons. Broadly speaking, the term inheritance-based polymorphism includes both class-based and interface-based polymorphisms.

> ℹ Note that the inheritance-based polymorphism, in the narrow sense, is really "overrated". In practice, the interfaces are much more versatile. Implementation reuse can be also done via "composition" without using inheritance.
>
> In the modern C#, regardless of whether it is a good thing or not, even an interface can include implementations. (They are called the "default implementations" because they provide the default implementations for the concrete types that inherit from the interface.)

22.2. Code Review - Total Salary Calculator

The sample code of this lesson demonstrates a few salient aspects of the inheritance-based polymorphism. It should be noted that the sample code is not a real-world example.

The main program creates a few "Organizations" with a number of "Employees" and compute their "total expenses".

salary-calculation/Program.cs (lines 3-32)

```
var managerOrg1 = new Management("M1") {
    new Manager("M1a"),
    new Manager("M1b"),
};
var workerOrg1 = new RankAndFile("W1") {
 8     new Worker("W1m"),
 9     new Worker("W1n"),
10 };
11 var workerOrg2 = new RankAndFile("W2") {
12     new Worker("W2x"),
13     new Worker("W2y"),
14     new Worker("W2z"),
15 };
16
17 var allOrgs = new List<Organization>() {
18     managerOrg1,
19     workerOrg1,
20     workerOrg2,
21 };
22
23 var allEmployees = new List<Employee>();
24 allOrgs.ForEach(o => o.GetAllEmployees().ForEach(e =>
   allEmployees.Add(e)));
25
26 var total1 = 0m;
27 allEmployees.ForEach(e => total1 += e.Salary);
28 Console.WriteLine($"total1 = {total1}");
29
30 var total2 = 0m;
31 allOrgs.ForEach(o => total2 += o.TotalExpense);
32 Console.WriteLine($"total2 = {total2}");
```

Note that the `allOrgs` list is declared to be a list of `Organizations` although their elements are of types, `Management` or `RankAndFile`. Likewise, the `allEmployees` list, explicitly declared as a list of `Employees`, contains the items that are of types, `Manager` or `Worker`.

In this example, the `allEmployees` list is built using the `ForEach()` method defined in `List<T>` from the `System.Collections.Generic` namespace (line 24), which takes a delegate argument. In this statement, both `ForEach()` methods take " Lambda expressions" as arguments, as we will discuss in more detail shortly.

Note that the total expense calculations (lines 27 and 31) both use the `ForEach()` method.

> `System.Collections.Generic` is also one of the special system namespaces as of C# 10.0. You neither have to fully qualify the names from these namespaces nor have to use the explicit `using` declarations.

As alluded, there are two kinds of `Employees` in this "Corporation" (namespace), namely, `Workers` and `Managers`.

salary-calculation/Employees.cs (lines 3-17)

```
 3 public abstract record Employee(string Name) {
 4     public abstract decimal Salary { get; }
 5 }
 6
 7 public sealed record Manager(string Name) : Employee(Name) {
 8     public override decimal Salary => 1_000_000m;
 9
10     public string Manage() => $"I just manage and I get paid {Salary} dollars.";
11 }
12
13 public sealed record Worker(string Name) : Employee(Name) {
14     public override decimal Salary => 100_000m;
15
16     public string Work() => $"I work and I get paid {Salary} dollars for my hard work.";
17 }
```

Interestingly, all `Managers` in this `Corporation` make a million dollars annually, whereas all rank and file `Workers` have an annual salary of only 100,000 dollars. ☺

As we will discuss shortly, a `Worker` *is* an `Employee` in this example. Likewise, a `Manager` *is* an `Employee` as well.

There are also two kinds of `Organizations` in this `Corporation`, a `Management`, which includes only `Managers`, and a `RankAndFile`, which includes only `Workers`. (As stated, this is a somewhat contrived example for more than one reason. 😀)

salary-calculation/Organizations.cs (lines 3-21)

```
public abstract class Organization : List<Employee> {
    public string Name { get; }

    protected Organization(string name) => Name = name;

    public decimal TotalExpense {
        get {
            var sum = 0m;
            ForEach(s => sum += s.Salary);
            return sum;
        }
    }

    public List<Employee> GetAllEmployees() => new(this);

    public Employee? GetEmployee(string name) =>
GetEmployeeByName(name);

    protected virtual Employee? GetEmployeeByName(string name) =>
null;
}
```

salary-calculation/Organizations.cs (lines 23-34)

```
public sealed class Management : Organization {
    public Management(string name) : base(name) { }

    protected override Manager? GetEmployeeByName(string name) {
        foreach (var w in this) {
            if (w.Name == name) {
                return w as Manager;
            }
        }
        return null;
    }
}
```

salary-calculation/Organizations.cs (lines 36-47)

```
public sealed class RankAndFile : Organization {
    public RankAndFile(string name) : base(name) { }

    protected override Worker? GetEmployeeByName(string name) {
        foreach (var w in this) {
            if (w.Name == name) {
                return w as Worker;
            }
        }
        return null;
    }
}
```

The `TotalExpense` of the `Corporation` is the sum of the salaries of all `Employees`.

22.3. Pair Programming - OOP, Inheritance, Polymorphism, Nullable Reference Types, `ForEach`

A `class` in C# can inherit from another `class`. Likewise, a `record` type can inherit from another `record` type.

Every concrete type inherits from `object`, ultimately. And, no concrete type can inherit from more than one concrete type in C#.

Hence, a user-defined type can have a linear inheritance hierarchy that eventually reaches the top, `object`. (Or, the root, depending on the how you look at it.)

In this example, an `Employee` record, which implicitly inherits from `object`, is declared as an `abstract record`. An `Employee` object cannot be directly instantiated. It only defines a "type", like an interface.

Instead, two other concrete types, `Manager` and `Worker`, inherit from this abstract record, which can be instantiated.

The colon (`:`) syntax, e.g., `record Manager(string Name) : Employee(Name)`, indicates that the record being declared, e.g., `Manager`, is a subtype of the record on the right-hand side, e.g., `Employee` in this case.

Note the syntax. `Manager` uses the positional record syntax, and more or less the same constructor syntax is used for its base class (or, base record) declaration.

Note that the abstract record type `Employee` includes a read-only property, `Name`, through the positional syntax, and in addition, it includes another `abstract` readonly property, `Salary`.

Abstract properties and methods, which do not include definitions, can only be declared in abstract classes and records.

Abstract properties and methods can be "overridden" in its sub- classes and records by declaring the properties or methods with the same signature and using the `override` keyword.

For instance, both `Manager` and `Worker` records provide their implementations for `Salary` by overriding the abstract property:

```
public override decimal Salary => 1m;
```

> As you will remember, the suffix `m`, or `M`, indicates that the number is a decimal literal. `1M` does not mean that it is one million! 😀

`Salary` is an auto-property (just like `Name`), and hence it includes an (implicit) implementation (e.g., with the backing field). However, it is declared as `abstract`, forcing the child records to explicitly implement this property.

> Furthermore, the `Salary` property, or its backing field, cannot be initialized as the `Salary` record is currently defined. Why? What change(s) need to be made to make `Salary` non-abstract?

Note that the overriding methods or properties cannot be declared with broader access levels.

Just to recap, there are two kinds of access modifiers for the top-level types, `public` and `internal`. The `internal` access level is the default, and if no access modifier is specified, then the type's access level is `internal`, meaning that the type is only accessible within the same assembly. The `public` types can be accessed from other assemblies (as long as they can access this assembly).

For the "nested types" and the type members, there are actually six different access levels.

public

Public names (members and nested types) can be accessed outside the type definition.

protected internal

Conceptually, `protected` OR `internal`. That is, the `protected internal` names are accessible from an inherited type or any type within the same assembly.

protected

Protected names can be accessed from the type's child/inherited types, either within or outside the assembly.

internal

Internal names are accessible within the same assembly.

private protected

Conceptually, `protected` AND `internal`. That is, the `private protected` names are accessible from an inherited type within the same assembly.

private

Private names cannot be accessed from outside the type definition.

These access modifiers are listed here more or less in the descending order, from "more open" to "more closed".

When you "override" a member method or property, you can either use the same access modifier or you can use a "more restricted" one as long as it does not change the parent type's public API. That is, the `public` methods and properties in the super class/record can be only be overridden as `public`.

Note that we do not discuss nested types and functions in this book. As stated, this is an introductory book and our goal is not to

> go through all the features of C#. The readers are encouraged to learn more about these concepts, if interested, from other resources.

In this example, the `Manager` record includes a method that is not in its base class, namely, `Manage()`. Likewise, the `Worker` record includes a method that is not in its base class `Employee`, namely, `Work()`.

The implementations of these two methods are pretty "lame". ☺ They merely return (different) strings. But, nonetheless, they illustrate an important concept in the object oriented programming.

The `Manage()` method uses the property `Salary`, and it uses the overridden Manager implementation, `Manager.Salary`. Likewise, the `Work()` method uses the `Salary` implementation overridden in the `Worker` record. This holds true even when we use the variables of type `Employee`, as we will further discuss in this lesson.

Now turning to the "Organizations", we have a similar type inheritance structure on the `Organization` side.

`Organization` is an `abstract class`. It includes a readonly property, `Name`, and it defines a protected constructor. `Protected` constructors, fields, properties, and methods, can be used, or overridden, only in its subclasses, or "subrecords".

As before, the colon (`:`) indicates the parent - child relationship in the class declaration. For example, `class Management : Organization` means that the `Management` class inherits from the `Organization` class. Likewise, `RankAndFile` also inherits from `Organization`.

In the custom type hierarchy, the constructors of a child type must call one of the constructors of its parent type. We have seen the special syntax used in the case of (positional) records.

In the case of `classes`, we use the `base` keyword. For instance, for both `Management` and `RankAndFile`, we explicitly "call" `base(name)` (at least syntactically, after the colon), which corresponds to the protected `Organization(string name)` constructor.

This explicit syntax for "calling" a base class constructor can be omitted if we are calling its no-argument default constructor. Omitting the `base(…)` call in cases where the base class does not have a default constructor will lead to a compile error. (For instance, the `Organization` class has no default constructor.)

Notice, in this example, that the `Employee? GetEmployeeName(string)` protected method is declared as `virtual`, and it includes an implementation (unlike `abstract` methods or properties).

`Virtual`, as well as `abstract`, properties and methods in a class/record can be overridden in its subtypes, using the `override` keyword. (For obvious reasons, virtual or abstract methods and properties cannot be `private`. Otherwise, they cannot be overridden from the child types.)

Depending on the actual concrete type used, at runtime, appropriate methods/properties are called.

The `Organization` class in this sample code includes a separate method `GetEmployee()`, in addition to `GetEmployeeByName()`, for demonstration purposes. (This division might not have been necessary in practice.)

The `GetEmployee()` method is not `virtual`, but it calls the virtual method, `GetEmployeeByName()`.

```
public Employee? GetEmployee(string name) => GetEmployeeByName(name);
```

Although this implementation appears to call `Organization.GetEmployeeByName()`, that interpretation is not entirely correct.

Since `GetEmployeeByName()` is a virtual method, depending on the actual type of a variable (of type `Organization`) at runtime, different `GetEmployeeByName()` methods will be called. That is, at runtime, either `Management.GetEmployeeByName()` or `RankAndFile.GetEmployeeByName()` will be called.

(Note that an instance of `Organization` cannot be directly created since it is an abstract type.)

> This can be a difficult concept to understand for people new to the object oriented programming styles, but as we emphasize, it will seem natural after enough exposure to these concepts. The question is not about "understanding". It is about "getting familiar" with them.
>
> The whole machinery of "abstract/virtual" and "override" is there to support this kind of polymorphic behavior. To repeat, for the abstract and virtual methods and properties, depending on the runtime type of an object (e.g., based on the actual constructor used to create the object), the overridden methods and properties, if any, may be used instead of those defined in its parent/base types regardless of how the object's type is declared in the source code.

In this example, it should be noted that this particular scheme (e.g., relying on the polymorphic behavior) might not have worked if we declared `GetEmployeeName()` as `abstract` rather than `virtual`. Abstract methods and properties can only used in other abstract methods or properties.

One interesting thing about this `GetEmployeeName()` "virtual method" is that the overriding methods have slightly different method signatures.

For `Organization`,

```
virtual Employee? GetEmployeeName(string) { }
```

For `Management`,

```
override Manager? GetEmployeeName(string) { }
```

For `RankAndFile`,

```
override Worker? GetEmployeeName(string) { }
```

This is allowed in C# because `Manager` inherits from `Employee`, and `Worker` inherits from `Employee`. You can return a subtype of the type that is returned in the parent's virtual method/property. This is called the "covariance returns".

Normally, overriding methods and properties in a child class/record should have exactly the same signatures. The "covariance returns" is an exception to this rule.

In the main program, we create Employee objects and Organization objects.

They are statically declared as `Employees` and `Organizations`.

```
var allEmployees = new List<Employee>();
```

```
var allOrgs = new List<Organization>();
```

And yet, at runtime, the correct implementation records/classes are used and the proper overridden properties and methods are called.

For example, the correct `Salary` value is used for `Managers` and `Workers` in the "total expense" calculations.

Even if we declare them more explicitly, there is no difference. For example,

```
Employee m1a = new Manager("M1a");
Employee m1b = new Manager("M1b");
Employee w1m = new Worker("W1m");
Employee w1n = new Worker("W1n");
Employee w2x = new Worker("W2x");
Employee w2y = new Worker("W2y");
Employee w2z = new Worker("W2z");
Organization managerOrg1 = new Management("M1") { m1a, m1b };
Organization workerOrg1 = new RankAndFile("W1") { w1m, w1n };
Organization workerOrg2 = new RankAndFile("W2") { w2x, w2y, w2z };
```

Although all these objects are statically declared with their base types, `Employee` and `Organization`, respectively, the correct types are used at runtime.

In a sense, this is the essence of the polymorphism. The variable, `m1a` for instance, is declared as a type `Employee` in a source program, but it is more specifically a variable of a subtype `Manager` since it is crated as `Manager` at runtime.

Likewise, `w1m` of type `Employee`, for instance, is indeed a variable of subtype `Worker`, more specifically.

The variables `m1a` and `w1m` are, on the one hand, of the same type `Employee`, but on the other hand, they can behave differently (e.g., when we are calling `GetEmployee()`) because they are instances of different subtypes.

For instance, `managerOrg1[0].GetType()` (from `object.GetType()`) will return the more specific type `Corporation,Manager` rather than a parent `Corporation.Employee` (although `managerOrg1[0]` is declared as `Employee`).

C#, and .NET, includes a number of operators to help manage types in a program. For example, the `is` operator can be used to test if an object is of a given type (or, one of the types in the type inheritance hierarchy).

```
Console.WriteLine($"{managerOrg1[0] is Employee}");
Console.WriteLine($"{managerOrg1[0] is Manager}");
```

Both statements print out `true` whereas `managerOrg1[0] is Worker` evaluates to `false`. (We can use the `m1a` variable instead of `managerOrg1[0]` with the same results in this illustration.)

Note that the value of an expression `v is object`, for any non-null variable `v`, is always `true` regardless of the type of `v` since every type inherits from the base type `object` in C#.

Another useful type-related operator is a cast operator `as`. It casts a variable of a reference type to a specified type. If the specified type is not compatible with the variable, that is, if the variable is not of the given type (in the type inheritance hierarchy), then it evaluates to `null`.

The expression `managerOrg1[0] as Manager` will successfully cast `managerOrg1[0]` to the `Manager` type whereas `managerOrg1[0] as Worker` will result in `null`.

The modern C# allows specifying a name of the cast variable in the `is` expressions. As an example,

```
Console.WriteLine($"{((managerOrg1[0] is Manager mgr) ? mgr.Manage()
: "I am not a manager")}");
Console.WriteLine($"{((managerOrg1[0] is Worker wkr) ? wkr.Work() :
"I am not a worker")}");
```

These two statements will output the following:

```
I just manage and I get paid 1000000 dollars.
I am not a worker
```

It should be noted that the base type `Employee` includes neither `Manage()` nor `Work()` methods. Only an object of the `Manager` type can use the method `Manage()` and only an object of the `Worker` type can use the method `Work()`. That is, `managerOrg1[0].Manage()` or `managerOrg1[0].Work()` would have led to a compile error at build time.

This syntax saves us from the need to do an explicit casting after the `is` testing.

In the case of the `as` expressions, we can do something like this:

```
Console.WriteLine($"{(managerOrg1[0] as Manager)!.Manage()}");
Console.WriteLine($"{(managerOrg1[0] as Worker)?.Work()}");
```

The first `WriteLine()` method will print out

```
I just manage and I get paid 1000000 dollars.
```

On the other hand, the second `WriteLine()` method will print nothing but a newline (because the argument evaluates to an empty string).

Since the `as` expression can result in `null`, we will need to do a null check. In this particular case, since we know that the runtime type of `managerOrg1[0]` is `Manager` (and that the expression cannot result in `null`), we do a little cheating and use the `!` operator.

It is an explicit notice to the compiler that since we know what we are doing you can ignore this expression from your null checks. (`!` is called the "null forgiving` operator", or the "null suppression operator".)

On the other hand, in the case of the second statement, we use the "null conditional operator", `?`. The `Work()` method will be only called if the `as` expression (before the `?`) is non-null. If the `as` expression evaluates to `null` (because `managerOrg1[0]` cannot be cast to `Worker`), then the overall value of the entire expression, `(managerOrg1[0] as Worker)?.Work()`, is `null`.

One thing to note is that the `Management` and `RankAndFile` classes use the " collection initialization syntax". For example,

```
var managerOrg1 = new Management("M1") {
    new Manager("M1a"),
    new Manager("M1b"),
};
```

This is possible because its base class, `Organization`, inherits from a collection type, `List<Employee>`. Hence, these classes are effectively collection types (through inheritance).

The main program also uses a "higher order" method, `ForEach()`, which is defined as a member method on `List<T>`.

For instance,

```
allEmployees.ForEach(e => total1 += e.Salary);
```

The type of the argument of `ForEach()` is a delegate. More specifically, `e ⇒ total1 += e.Salary` is a "Lambda expression", which is a sort of a literal of a delegate type.

In this particular case, the argument of `ForEach()` is of type `Action<Employee>`. The type `Action<T>`, from the `system` namespace, encapsulates a method that has a single parameter and does not return a value.

A Lambda expression is like an anonymous function. Note the syntax. The variable before the fat arrow (`⇒`) is an argument. If there is more than one arguments, we put them in a pair of parentheses. If there is only one argument, the parentheses are typically omitted.

The expression after the fat arrow is the body of the function, an expression `total1 += e.Salary` in this case.

The `ForEach()` method evaluates the lambda expression for all elements of the list, `allEmployees`, in this case.

This statement is in fact equivalent to the following:

```
foreach (var e in allEmployees) {
    total1 += e.Salary;
}
```

Although `ForEach()` is not a LINQ method, technically, it follows the same pattern as many of the LINQ methods. We will show example usages of some LINQ extension methods later in the book.

Pop quiz. ☺

The constructor methods cannot be `virtual`. Why do you think that is? Likewise, constructors cannot be `abstract`. Would it make sense to have an `abstract` constructor for a type?

Summary

We learned the basics of inheritance in C#. Virtual and abstract methods and properties of a type can be overridden from its subtypes, records and classes.

In particular, we used the keywords `abstract`, `virtual`, and `override`, among other things. Objects of a reference type (e.g., class, struct, and interface) can exhibit polymorphic behavior at runtime.

We also covered the six access modifiers used to control the access levels of the member methods and properties of a type. Namely, `public`, `internal`, `protected`, `protected internal`, `private protected`, and `private`.

C# provides a number of operators for checking types and casting objects between different types. We explored the some basics of the `is` and `as` operators. Although we did not cover in this lesson, C# also has an operator, `typeof`, to get the `System.Type` instance for a given type name. The `object.GetType()` method, available in all objects, can also be used to get the type of an object.

We also learned basic usages of the functional style methods (e.g., `List<T>.ForEach()`) and the lambda expressions. The `List<T>.ForEach()` method, as well as many of the LINQ extension methods, takes a delegate as one of its arguments.

Chapter 23. Areas and Perimeters

23.1. Introduction

We will explore, in this lesson, how the interface multiple inheritance works in C#.

> This lesson is going to be the most boring one in this book. 😀 You can skip this lesson if you are not interested in all the nitty gritty details of the C# grammar. C# is a rather complex language and in order to become proficient, however, you will at least need to be aware of certain concepts although you may not have to remember the details.

23.2. Code Review - Area and Perimeter Calculator for Multiple Shapes

Let’s start with simpler examples. How does the interface (single) inheritance works in C#.

> "Interface inheritance" is not the most commonly used term in this context. We more commonly say that *a class "implements" an interface(s),* etc. But, as long as we know what we are talking about here, what kind of terms we use is not that critical.

The complexity comes from the fact that C# gives us too many options. The choices are blessings and curses at the same time.

When we define an interface, we can optionally provide a default implementation for a method or property. When we implement an interface method or property, a class can either *explicitly* or *implicitly* implement the method or property. And so forth.

In situations like this, we often face what is called the "combinatorial explosion".

Let’s take a look at some concrete examples. Here’s a simple interface:

area-calculation/si1/Area.cs (lines 4-10)

```
public interface IArea {
    /// <summary>
    /// Returns the area.
    /// This method needs to be implemented in the concrete types.
    /// </summary>
    float Size();
}
```

We will discuss more on the "XML doc comments" later. For now, the interface `IArea` declares one method `float Size()`. This interface method is implicitly `abstract`, and it requires an implementation in the concrete types. This method is also implicitly `public`.

We create three types that implement this `IArea` interface, in the `SI1` namespace (although it is not explicitly shown in the code snippets).

The `SI1.ShapeB` class "implicitly" implements the `Size()` method:

area-calculation/si1/Shapes.cs (lines 4-9)

```
public class ShapeB : IArea {
    /// <summary>Implicit/normal implementation of IArea.Size().</summary>
    public float Size() {
        return 5.0f;
    }
}
```

This is the most common, and traditional, way of implementing an interface method or property. Let’s test it with a simple program:

```
ShapeB shape1B = new();
var size1B = shape1B.Size();
Console.WriteLine($"size1B = {size1B}");
```

This outputs, as expected,

```
size1B = 5
```

Not much surprise here. Now, let's try the "explicit implementation":

area-calculation/si1/Shapes.cs (lines 12-17)

```
public class ShapeC : IArea {
    /// <summary>Explicit implementation of IArea.Size().</summary>
    float IArea.Size() {
        return 10.0f;
    }
}
```

Note the syntax, `float IArea.Size() { }`.

Through this explicit implementation syntax, the class `ShapeC` fulfills its duty, so to speak, as a type that can behave like `IArea`. However, the `Size()` method cannot be directly called on an object of a type `ShapeC`. It can only be called on a type `IArea` (as statically declared).

For example,

```
ShapeC shape1C = new();
var size1C = (shape1C as IArea).Size();
Console.WriteLine($"size1C = {size1C}");
```

The `as` casting, `shape1C as IArea`, makes the compiler happy, and we can now call the `Size()` method on the `shape1C` object. The output is as expected:

```
size1C = 10
```

Alternatively, we can declare the `ShapeC` variable as `SI1.IArea`, and we can call the explicitly defined `Size()` method. The output will be the same as before.

```
IArea shape1Ca = new ShapeC();
var size1Ca = shape1Ca.Size();
Console.WriteLine($"size1Ca = {size1Ca}");
```

What happens if we implement the `Size()` method both implicitly and explicitly?

area-calculation/si1/Shapes.cs (lines 23-33)

```
public class ShapeD : IArea {
    /// <summary>It has nothing to do with IArea.Size().</summary>
    public float Size() {
        return 15.0f;
    }

    /// <summary>Explicit implementation of IArea.Size().</summary>
    float IArea.Size() {
        return 20.0f;
    }
}
```

If we use the explicit type `ShapeD` (at build time) for a `ShapeD` object, then the call `shape1D.Size()` ends up using the implementation of the concrete type:

```
ShapeD shape1D = new();
var size1D = shape1D.Size();
Console.WriteLine($"size1D = {size1D}");
```

```
size1D = 15
```

If, on the other hand, we call `Size()` on the variable of a type `IArea` (at build time), then it ends up using the explicit implementation `IArea.Size()`:

```
IArea shape1Da = new ShapeD();
var size1Da = shape1Da.Size();
Console.WriteLine($"size1Da = {size1Da}");
```

```
size1Da = 20
```

Is that confusing enough for you? 😀 If you are still here, then let's make things slightly more confusing. 😊

An interface method can have a default implementation. We have just seen the interplay between the implicit and explicit implementations of an "abstract" interface method. Now let's see what happens when we add a default implementation for an interface method.

Here's a new interface:

area-calculation/si2/Area.cs (lines 4-10)

```
public interface IArea {
    /// <summary>
    /// Returns the area.
    /// Has a default implementation.
    /// </summary>
    float Size() => 30.0f;
}
```

This interface method has a default implementation. In this example, it simply returns a fixed number `30.0`.

We create four different types that implement this new `IArea` interface (in the `SI2` namespace).

The `SI2.ShapeA` class does not include its own implementation of the `Size()` method. It instead relies on `SI2.IArea` for the method implementation.

area-calculation/si2/Shapes.cs (line 4)

```
4 public class ShapeA : IArea { }
```

Although it technically "implements" the interface `IArea` (as indicated by the public declaration `ShapeA : IArea`), we cannot call the `Size()` method directly on an object of type `ShapeA`.

Instead, we need to explicitly use the interface type `IArea`.

```
IArea shape2Aa = new ShapeA();
var size2Aa = shape2Aa.Size();
Console.WriteLine($"size2Aa = {size2Aa}");
```

This prints out

```
size2Aa = 30
```

As expected, it uses the default implementation of `IArea.Size()` (as evidenced by the output `30`).

Next, let’s try the "normal" way of implementing an interface method:

area-calculation/si2/Shapes.cs (lines 7-10)

```
public class ShapeB : IArea {
    /// <summary>An implicit/normal implementation of IArea.Size().</summary>
    public float Size() => 35.0f;
}
```

Let’s test this with a simple program:

```
var shape2B = new ShapeB();
var size2B = shape2B.Size();
Console.WriteLine($"size2B = {size2B}");
```

It behaves the same way as before (`SI1.ShapeB`):

```
size2B = 35
```

Since the class `ShapeB` includes the "normal" implementation of `IArea.Size()`, the interface’s default implementation is irrelevant in this case. This is true even if you declare a variable of type `ShapeB` explicitly as `IArea`.

```
IArea shape2Ba = new ShapeB();
var size2Ba = shape2Ba.Size();
Console.WriteLine($"size2Ba = {size2Ba}");
```

This prints out

```
size2Ba = 35
```

That is, the default interface implementation does not "overwrite" the implementation of the concrete type. This has an (important) implication that even if you add a new default implementation to a method in an existing interface the behavior of any existing types that implement the interface do not change with regards to that method.

What if we explicitly implement `IArea.Size()` in our type?

area-calculation/si2/Shapes.cs (lines 13-16)

```
public class ShapeC : IArea {
    /// <summary>An explicit implementation of IArea.Size().</summary>
    float IArea.Size() => 40.0f;
}
```

The class `SI2.ShapeC` again fulfills its "contractual obligation" (e.g. `ShapeC : IArea`) by explicitly implementing `IArea.Size()`. However, as with `SI1.ShapeC` before, we cannot call the `Size()` method on an object that is statically declared as `Shape2C`.

Instead, we will have to use a variable of the interface type `IArea` to be able to use the `Size()` method.

```
IArea shape2Ca = new ShapeC();
var size2Ca = shape2Ca.Size();
Console.WriteLine($"size2Ca = {size2Ca}");
```

This program prints out

```
size2Ca = 40
```

Note that whether the method has a default implementation or not is of no significance in this case since we explicitly implement the method.

As in the previous examples (from the namespace `SI1`), adding the "same" method to the type that already includes an explicit implementation of the interface method does not alter the object's behavior with regards to that particular interface.

area-calculation/si2/Shapes.cs (lines 22-28)

```
public class ShapeD : IArea {
    /// <summary>It has nothing to do with IArea.Size().</summary>
    public float Size() => 45.0f;

    /// <summary>An explicit implementation of IArea.Size().</summary>
    float IArea.Size() => 50.0f;
}
```

If we use the explicit type `ShapeD` (at build time) for a `ShapeD` object, then the call `shape1D.Size()` ends up using this implementation:

```
var shape2D = new ShapeD();
var size2D = shape2D.Size();
Console.WriteLine($"size2D = {size2D}");
```

```
size2D = 45
```

If we call `Size()` on the variable that is statically declared as `SI2.IArea`, on the other hand, then it ends up using the explicit implementation `IArea.Size()`:

```
IArea shape2Da = new ShapeD();
var size2Da = shape2Da.Size();
Console.WriteLine($"size2Da = {size2Da}");
```

```
size2Da = 50
```

Again, this behavior is no different regardless of whether we have an interface default implementation or not.

> What is "wrong" with this picture?
>
> The separation of "interface" and "implementation" is a fundamental tenet in programming. C# violates this fundamental tenet by introducing this concept of implicit vs explicit interface method/property implementations and the interface method/property default implementations.
>
> Whether a type implements an interface method/property explicitly or not is an implementation detail. And yet, as we have seen in some examples, the usage of an object depends on how the interface method/property is implemented.
>
> Whether an interface method/property has a default implementation or not is an implementation detail. And yet, again, as we have seen in some examples, the usage of an object depends on whether an interface method/property has a default implementation or not.
>
> The author does not recommend the use of interface

> method/property default implementations. There are use cases for this, and whether those are valid, or good, use cases or not is debateable. If you need to share an implementation across multiple types, then consider using (abstract) base types rather than the interfaces with default implementations.
>
> As for the interface method/property explicit implementations, there are some limited use cases when a type inherits from multiple interfaces, as we will see next.

OK. You are still here. That means we can go a little bit deeper. Let's make things just a tad bit more complicated. 😀

A type can implement multiple interfaces. Normally, it works just like having multiple "single inheritances". ☺ The complication arises when two or more interfaces declare methods/properties with the same name.

The following is another somewhat convoluted example, whose main purpose is to illustrate the multiple interface inheritance feature in C#.

The following two interfaces (in the `MI1` namespace) both happen to declare the method `float Size()`.

area-calculation/mi1/Area.cs (lines 4-10)

```
public interface IArea {
    /// <summary>
    /// Returns the area.
    /// This method needs to be implemented in the concrete types.
    /// </summary>
    float Size();
}
```

area-calculation/mi1/Perimeter.cs (lines 4-10)

```
public interface IPerimeter {
    /// <summary>
    /// Returns the perimeter.
    /// This method needs to be implemented in the concrete types.
    /// </summary>
    float Size();
}
```

In general, it is best to avoid the name conflicts like this if you can, but it is sometimes beyond your control. You may have to use different interfaces from multiple libraries, for instance.

Our `Shape` types implement both `IArea` and `IPerimeter` interfaces.

The simplest example, `MI1.ShapeB`, includes an implementation of the `Size()` method, which serves as a concrete implementation for both `IArea.Size()` and `IPerimeter.Size()`.

area-calculation/mi1/Shapes.cs (lines 4-7)

```
public class ShapeB : IArea, IPerimeter {
    /// <summary>An implicit/normal implementation for IArea.Size() and IPerimeter.Size().</summary>
    public float Size() => 100.0f;
}
```

Note that this may not always be feasible since the methods with the same name from different interfaces may (most likely) have different meanings.

In this example, areas and perimeters are two different concepts. It will be very unlikely that we can come up with a single implementation that satisfies the requirements of both methods.

Syntactically, however, this provides the cleanest implementation.

```
ShapeB shape1B = new();
var size1B = shape1B.Size();
Console.WriteLine($"size1B = {size1B}");

IArea shape1Ba = new ShapeB();
var size1Ba = shape1Ba.Size();
Console.WriteLine($"size1Ba = {size1Ba}");

IPerimeter shape1Bp = new ShapeB();
var size1Bp = shape1Bp.Size();
Console.WriteLine($"size1Bp = {size1Bp}");
```

Calling the `Size()` method on three different objects, which have statically different types, will yield the same result since they all end up calling the same (one) implementation.

```
size1B = 100
size1Ba = 100
size1Bp = 100
```

Unfortunately, as stated, this is a rare case in practice.

Typically, in situations like this, a common solution is to implement the two interface methods explicitly. For instance, the `ShapeC` class does that:

area-calculation/mi1/Shapes.cs (lines 13-19)

```
13 public class ShapeC : IArea, IPerimeter {
    /// <summary>An explicit implementation of IArea.Size().</summary>
    float IArea.Size() => 110.0f;

    /// <summary>An explicit implementation of IPerimeter.Size().</summary>
    float IPerimeter.Size() => 120.0f;
}
```

One downside of the interface method explicit implementation is, as we have discussed earlier, that you can no longer use the type, `ShapeC` in this case, directly to access the interface methods. Either you need to cast an object to a particular interface (e.g., using the `as` cast) or declare it as an interface in the first place.

```
IArea shape1Ca = new ShapeC();
var size1Ca = shape1Ca.Size();
Console.WriteLine($"size1Ca = {size1Ca}");

IPerimeter shape1Cp = new ShapeC();
var size1Cp = shape1Cp.Size();
Console.WriteLine($"size1Cp = {size1Cp}");
```

The above test code yields the following output, as expected:

```
size1Ca = 110
size1Cp = 120
```

As before, once the methods from both interfaces are explicitly implemented, any method in a concrete type that has the same name and signature provides no polymorphic behavior.

area-calculation/mi1/Shapes.cs (lines 26-35)

```
public class ShapeD : IArea, IPerimeter {
    /// <summary>It has nothing to do with IArea.Size() or IPerimeter.Size().</summary>
    public float Size() => 130.0f;

    /// <summary>An explicit implementation of IArea.Size().</summary>
    float IArea.Size() => 140.0f;

    /// <summary>An explicit implementation of IPerimeter.Size().</summary>
    float IPerimeter.Size() => 150.0f;
}
```

```
ShapeD shape1D = new();
var size1D = shape1D.Size();
Console.WriteLine($"size1D = {size1D}");
```

The `WriteLine()` method prints out this:

```
size1D = 130
```

For this type, `ShapeG`, only the `IArea.Size()` method is explicitly implemented. The class includes the normal/implicit implementation of `Size()`.

area-calculation/mi1/Shapes.cs (lines 42-48)

```
public class ShapeG : IArea, IPerimeter {
    /// <summary>An implicit/normal implementation of
IPerimeter.Size().</summary>
    public float Size() => 160.0f;

    /// <summary>An explicit implementation of
IArea.Size().</summary>
    float IArea.Size() => 170.0f;
}
```

If you test various usages of objects of the `ShapeG` type,

```
ShapeG shape1G = new();
var size1G = shape1G.Size();
Console.WriteLine($"size1G = {size1G}");

IArea shape1Ga = new ShapeG();
var size1Ga = shape1Ga.Size();
Console.WriteLine($"size1Ga = {size1Ga}");

IPerimeter shape1Gp = new ShapeG();
var size1Gp = shape1Gp.Size();
Console.WriteLine($"size1Gp = {size1Gp}");
```

You get the result as follows:

```
size1G = 160
size1Ga = 170
size1Gp = 160
```

The implicitly defined method `Size()` provides a polymorphic behavior for `ShapeG` and `IPerimeter`, whereas the `IArea.Size()` implementation is used only when an object is explicitly declared as `IArea`.

The opposite behavior is expected when we explicitly implement `IPerimeter.Size()` and provide a normal implementation for the `Size()` method. This is illustrated as `ShapeH`, as included in the full code listing in the appendix.

The full code listing also includes another example, in the namespace `MI2`, where the interfaces include default implementations for the `Size()` method. The behavior of various classes implementing the two interfaces can be more or less "predicted" based on what we have learned so far. We will leave it to the readers to verify their "intuitions".

So, what is the takeaways of this lesson?

First, do not use the interface default implementations. When you create a new interface, do not even think about using default implementations. [simile] As stated, if you need to share some common implementations, consider using an (abstract) base class.

When you feel the impulse to add a method or property, with or without a default implementation, to an existing interface, don't! Just don't! 😀 Interfaces are supposed to be immutable. *No matter what.*

Second, there may be legit cases where you will have to implement an interface method explicitly. For example, when there is a name collision when implementing multiple interfaces, you can implement one method/property in the normal way, as in our examples of `Shape1G` and `Shape1H`, and explicitly implement the rest.

23.3. Pair Programming - XML Doc Comments

One last thing we need to learn in this lesson is the XML documentation feature of C#.

This is originally based on JavaDoc, as is the case with many other programming languages, and it allows us to embed documentations in the code, within comments. This is called the "doc comment".

We can enable this feature in the project XML file by setting an attribute (within a `<PropertyGroup>`) as follows:

```
<GenerateDocumentationFile>true</GenerateDocumentationFile>
```

If you use the dotnet CLI tool, then the documentation, in XML, will be automatically generated when you run the `dotnet build` or `dotnet publish` commands. The default doc file name is the project name with the "xml" file extension, and it is created in the build directory by default.

To add a doc, you use the three-slash `///` comments in C#.

You can refer to the official XML doc reference page for more information: XML documentation comments [https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/xmldoc/].

In this lesson, we used only one XML tag `<summary></summary>`, primarily for demonstration. But, you can find other commonly used tags in this page: Recommended XML tags for C# documentation comments [https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/xmldoc/recommended-tags].

Here’s an example of the generated XML documentation:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>area-calculation</name>
    </assembly>
    <members>
        <member name="T:SI1.IArea">
            <summary>Area interface. Used to demonstrate an interface
inheritance.</summary>
        </member>
        <member name="M:SI1.IArea.Size">
            <summary>
            Returns the area.
            This method needs to be implemented in the concrete
types.
            </summary>
        </member>
        <member name="T:SI1.ShapeB">
            <summary>ShapeB illustrates a single interface
inheritance.</summary>
        </member>
        <member name="M:SI1.ShapeB.Size">
            <summary>Implicit/normal implementation of
IArea.Size().</summary>
        </member>
        <member name="T:SI1.ShapeC">
            <summary>ShapeC illustrates a single interface
inheritance.</summary>
        </member>
        <member name="M:SI1.ShapeC.SI1#IArea#Size">
            <summary>Explicit implementation of
IArea.Size().</summary>
        </member>
        <member name="T:SI1.ShapeD">
             <summary>
             ShapeD illustrates a single interface inheritance.
             It includes an explicit implementations of IArea.Size().
            </summary>
        </member>
        <member name="M:SI1.ShapeD.Size">
```

```
            <summary>It has nothing to do with
IArea.Size().</summary>
        </member>
        <member name="M:SI1.ShapeD.SI1#IArea#Size">
            <summary>Explicit implementation of
IArea.Size().</summary>
        </member>
        <member name="T:SI1.Driver">
            <summary>Test driver</summary>
        </member>
        <member name="M:SI1.Driver.M">
            <summary>Test Program</summary>
        </member>
    </members>
</doc>
```

One of the downsides of using XML comments in C#, in other JavaDoc style frameworks, is that it adds too much clutter to the source code. The XML syntax also adds some friction by making it a little bit more difficult for developers to quickly write the doc comments.

We do not use the XML doc comments, or other general code comments, in most of the example programs in this book. Let's hope that C# adopt a simpler, and easier to use, doc commenting scheme in the near future.

Summary

We explored the interface default implementation feature in this lesson. Our recommendation is not to use it if you can avoid it. ☺

In addition, we learned how to "explicitly" implement interface methods and properties. When there is a name conflict between multiple interfaces, we can explicitly implement some or all of them using the special "explicit implementation" syntax.

The best practice is to implement one (primary) method/property in the "normal" way, and implement the rest explicitly.

We also learned about the XML doc comments in C#. One can enable the doc generation by setting the `GenerateDocumentationFile` attribute to `true` in the XML project file.

Chapter 24. Simple Web Server

24.1. Introduction

(cs) is one of the most popular programming languages for Web development, especially on the backend side. "ASP.NET" is one of the most popular *cross-platform* Web service frameworks. Many C# developers use ASP.NET for web application development.

In this lesson, we will use a "low level" class, `HttpListener` from the `System.Net` namespace, to create a simple Web server.

As just stated, you will most likely use one of the "frameworks" to build Web apps, but this lesson covers some basics of HTTP programming.

Furthermore, we will explore the `async` - `await` programming style in C#.

24.2. Code Review - Async Web Server Program

The main program:

web-server-simple/Program.cs (lines 3-4)

```
var server = new Server();
await server.Run();
```

The main program is essentially a one-liner, `await new Web.Server().Run()`. It merely calls the `Run()` async method with `await`. When `await server.Run()` returns, the program terminates.

Here's the definition of the `Server` class.

web-server-simple/Server.cs (lines 6-31)

```
public sealed class Server {
    private readonly List<(string[], Func<HttpListenerContext, Task>)> handlers;

    public string TextMessage { get; init; } = "Hello World!";
    public object JsonPayload { get; init; } = new {
        Type = "Greeting",
        Message = "Hello World!",
    };

    public Server() {
        handlers = new() {
            (new[] { "http://127.0.0.1:8080/" }, (Func<HttpListenerContext, Task>)ProcessTextResponse),
            (new[] { "http://127.0.0.1:8080/json/" },
(Func<HttpListenerContext, Task>)ProcessJsonResponse),
        };
    }

    public async Task Run() {
        var tasks = new List<Task>();
        foreach (var (r, h) in handlers) {
            if (r.Length > 0) {
                var t = Listener.StartAsync(r, h);
                tasks.Add(t);
            }
        }
        await Task.WhenAll(tasks);
    }
```

web-server-simple/Server.cs (lines 33-44)

```
    private async Task ProcessTextResponse(HttpListenerContext
ctx) {
        byte[] data = Encoding.UTF8.GetBytes(TextMessage);
        ctx.Response.ContentLength64 = data.Length;

        using var stream = ctx.Response.OutputStream;
        stream.Write(data, 0, data.Length);
        stream.Flush();
        stream.Close();

        await Task.Delay(1);
        Console.WriteLine($"Responded at {DateTime.Now}");
    }
```

web-server-simple/Server.cs (lines 46-61)

```
    private async Task ProcessJsonResponse(HttpListenerContext ctx) {
        string json = JsonSerializer.Serialize(JsonPayload);
        byte[] data = Encoding.UTF8.GetBytes(json);

        var response = ctx.Response;
        response.ContentType = "application/json";
        response.ContentLength64 = data.Length;

        using var stream = response.OutputStream;
        stream.Write(data, 0, data.Length);
        stream.Flush();
        stream.Close();

        await Task.Delay(1);
        Console.WriteLine($"Responded at {DateTime.Now}");
    }
```

`Server` is a sealed class with the default constructor. What we are building here is clearly not a "data" type, and hence the `sealed class` should be our first choice.

A "server" is mainly about behavior. A server serves. It processes. It otherwise does something. A server is a `class`.

We use a small static method to encapsulate the logic of starting an instance of `System.Net.HttpLister`.

web-server-simple/Listener.cs (lines 4-19)

```
static class Listener {
    internal static async Task StartAsync(string[] prefixes, Func<HttpListenerContext, Task> handler) {
        using var listener = new HttpListener();
        foreach (string s in prefixes) {
            listener.Prefixes.Add(s);
        }

        listener.Start();
        Console.WriteLine($"Listening on {{{string.Join(", ", prefixes)}}}");

        while (true) {
            var context = await listener.GetContextAsync();
            await handler(context);
        }
    }
}
```

24.3. Pair Programming - Async Programming, `Func` Delegates, `HttpListener`, `JsonSerializer`

Because of the importance of Web programming, most programming languages and runtimes have a very good support for creating a simple Web server.

In C# and .NET, there is this class `HttpListener`, which provides all essential components to create a Web server.

The `Web.Server` class is a simple wrapper around `HttpListener`.

If you run a server instance, then it prints out some messages and "waits".

```
$ dotnet run
```

```
Listening on {http://127.0.0.1:8080/}
Listening on {http://127.0.0.1:8080/json/}
```

We can access the Web server by sending a request via Curl, for instance:

```
$ curl -i "http://127.0.0.1:8080/"

HTTP/1.1 200 OK
Server: Microsoft-NetCore/2.0
Date: Tue, 29 Jun 2021 04:51:18 GMT
Content-Length: 12

Hello World!
```

We can also use a Web browser. Here's another example:

```
$ curl -i "http://127.0.0.1:8080/json/hello"

HTTP/1.1 200 OK
Content-Type: application/json
Server: Microsoft-NetCore/2.0
Date: Tue, 29 Jun 2021 04:52:13 GMT
Content-Length: 44

{"Type":"Greeting","Message":"Hello World!"}
```

This simple Web server sends two different responses depending on the request URLs.

The `Server` class includes two init-only properties, `TextMessage` and `JsonPayload`, and they are initialized at the point of declarations. Notice the somewhat strange syntax to set the initial values for properties. The equal sign (`=`)

follows the getter/setter block ({ and }).

TextMessage is a string, whereas JsonPayload is declared merely as object. The JsonPayload property is initialized with an object of an "anonymous type".

In C#, you can create a (concrete) type using struct, class, or record. Then you can create an object of that type using new constructors or initializers.

There is another way. You can just create an object without explicitly specifying a particular type. The object is then given a specific type based on the initial value, but the type does not have a name. Hence, it is called an anonymous type.

It is really a one-off, one of a kind, type whose sole purpose is to give a type to the object that is being created without being given an explicit type.

Note the syntax. It is somewhat similar to the (typed) object initializer starting with new, but the type name is missing.

Anonymous types are more commonly used for local variables, which are declared with var because we do not have to specify an explicit type when we use the var declaration.

In this example, the usage is somewhat unusual, but it serves our purpose. We can set JsonPayload to a value of *any* type, including anonymous types.

> Note that the values of anonymous types are "pure data" objects. Anonymous types cannot include methods or implement interfaces, etc.

The Server class has the default constructor. And, we can set the values of these init-only properties using the object initializer syntax. For example,

```
var server = new Server() {
    TextMessage = "Hi",
    JsonPayload = new {
        Name = "Joe",
        Title = "Sir",
    }
};
```

In this case, `JsonPayload` is initialized with a value of a completely different (anonymous) type.

Note the (efficient) syntax of constructing the private readonly field, `handlers`, in the `Server` constructor:

```
handlers = new() {
    (new[] { "http://127.0.0.1:8080/" }, (Func<HttpListenerContext,
Task>)ProcessTextResponse),
    (new[] { "http://127.0.0.1:8080/json/" },
(Func<HttpListenerContext, Task>)ProcessJsonResponse),
};
```

All type names are missing from the `new` expressions. In "modern C#", you can literally save a few keystrokes if you know what you are doing. And, once you get used to it, this kind of syntax is much easier to read than having the explicit type names around.

The `handler` field is declared as a `List` of tuples, `(string[], Func<HttpListenerContext, Task>)`. Again, we do not (have to) create an explicit type for this pair. We simply use tuples.

The type of the first item is `string[]`. The type of the second item is `Func<HttpListenerContext, Task>`. This is a `delegate`, which we discussed earlier in the book.

The .NET standard library includes a number of predefined (generic) delegates. `Func<HttpListenerContext, Task>` is a delegate type that takes one argument of type `HttpListenerContext` and returns a value of a type `Task`.

As we will see shortly, `Task` is the fundamental building block of async, or asynchronous, programming in C#.

The `handlers` field stores pairs of "routes" and their "handler". In this sample code, the route "/" is mapped to the Server's instance method `ProcessTextResponse()`, and the route "/json/" is mapped to another method, `ProcessJsonResponse()`.

The `Listener.StartAsync()` static method takes these two values as arguments and sets up an `HttpListener` object to handle the requests targeted to the routes. `HttpListener` is an `IDisposable` and we use the `using` statement. An `HttpListener` can be used to serve multiple routes, and the "prefixes" is a collection (of strings).

Once the prefixes are set, we can "start" the `HttpListener` instance by calling the `Start()` method.

Then we process the requests through the `while (true)` infinite loop.

In the loop, we first get an `HttpListenerContext` object by calling `await listener.GetContextAsync()` and next call the handler delegate with that context object, ` await handler(context)`.

Note that `GetContextAsync()` is an async method and it has the suffix `Async` in the method name. It is a convention to use the "Async" suffix when there is a corresponding synchronous version of the method exists. For example, the `HttpListener` class has a (synchronous) method `GetContext()`. Hence the async version is named `GetContextAsync()`.

In some cases, we just use the "Async" suffix for any `async` method, but that is really a matter of style or preference.

In the sample code, we use Run() for one async method, and StartAsync() for another. Names are not that important as long as one follows the broad general conventions.

In general, async methods need to be called with await.

Any async method in C# returns, by definition, a type Task or Task<T> (for a particular type T). Methods that do not return any of these types are not async methods.

The Task (or, Task<T>) is very similar to "promises" or "futures" in other programming languages.

A synchronous, or blocking, method call does not return until the method completes its work. During that time, the caller cannot do anything else but wait. This can cause many issues such as non-responsiveness of UI, or under-utilization of the computer resources, etc.

An async, or non-blocking, method call, in contrast, returns immediately. It returns a Task object (Task or Task<T> for a type T), and it continues its work in a different thread.

The caller can check the status or the result of the async method call through the returned Task object.

But, the more common pattern is for the caller to await until the async method completes. But, unlike in the case of the synchronous calls, await does not block the current thread, which is the primary benefit of using the async methods.

C# is a procedural language at its core. Statements are supposed to run in a certain order. In general, a statement cannot, or should not, start before the previous one completes.

In the sample code, for instance,

```
var context = await listener.GetContextAsync();
await handler(context);
```

The program cannot call the `handler` delegate method (e.g., `server.ProcessTextResponse()` or `server.ProcessJsonResponse()`), for example, until it gets a context object from `listener.GetContextAsync()`. Logically, it has to be synchronous.

The beauty of the `async`/`await` paradigm is that the code is written as if they are synchronous calls (except for the uses of `async` and `await` keywords), and yet they are not "blocking". The program, or the thread, can do other things like accepting and processing user input, etc., while waiting for the previous async call to complete.

There is no need to use completely different syntaxes like "callbacks" or there is no need to explicitly create and manage separate threads, etc.

There are many different ways to do asynchronous programming in many different programming languages.

The C#'s async/await model is one of the best, the easiest, and the most intuitive to use (although it can be subjective).

The async/await programming model is based on a decades-old computer science theory, and now many programming languages are adopting this model, not just C#, to varying degrees, including modern Javascript/Typescript, Rust, Swift, Kotlin, etc., and yes now *even Python*, which has been around for over 30 years. (Their precise names and syntaxes vary.)

> ℹ In the modern programming using multi-core CPUs and what not, the need and importance of asynchronous programming is ever growing.

> Although we did not really discuss async/await until this lesson, that does not mean that it is less important than other topics we have covered earlier in this book. In order to become productive and effective, the readers will need to understand and use async/await programming well, including various APIs from the `Task` class, etc.
>
> When using third parity libraries and frameworks, such as ASP.NET, for example, many public APIs are `async` methods. Once you start really programming in C#, for any real tasks, you will realized that async/await is everywhere.

The `Server.Run()` method uses an interesting pattern.

It needs to call multiple aync methods (in a loop). Instead of calling each one with await sequentially, e.g., `await Listener.StartAsync(r, h)`, it first stores its returned object (of type `Task`) in a temporary list.

And then, after collecting all Tasks, it calls the static method `WhenAll()` of `Task`:

```
await Task.WhenAll(tasks);
```

This is an important async programming pattern to learn. There are a couple of things to note.

First, in this example, there is no need for one `Listener.StartAsync(r, h)` call to wait for another call to finish. They can run simultaneously. Hence calling them sequentially using `await` would have been rather inefficient. (On top of this, the `StartAsync()` method, in this particular example, does not return. It has an infinite loop.)

Next, we use the `WhenAll()` method. `WhenAll()` creates a task that will complete when all of the Task objects in the argument have completed. We create a new Task object based on other Tasks (e.g., returned from `StartAsync` calls) and we `await` the new Task to finish, which effectively making the program to `await` until all component tasks to finish.

The `Task` class provides other methods as well which implement different logic.

One important thing to notice here is that the `Server.Run()` method is declared as `async`.

```
public async Task Run() {
    // ...
}
```

You can only use `await` in an `async` method. As stated, you almost always use `await` for async methods, and therefore you can only use, or are most likely to use, async methods in another async method.

This is the case with `async Task Listener.StartAsync()` as well in this example.

Once you start using async APIs, you will end up creating more async methods of your own, and the users of your (public) APIs will end up using more async methods of their own, … And so on and so on.

Even our (implicit) `Main` method (in the Program.cs) is `async`. It calls an async method with `await`.

```
await server.Run();
```

Hence, it must be, and it is, an `async` method. (The compiler generates a version of the `Main()` methods that is compatible with the top-level statements, including the use `args`, etc.)

Async methods can return values by using a generic version of `Task`. For example, `HttpListener.GetContextAsync()` returns `Task<HttpListenerContext>`. When we call this method with `await`, the "real" return value is of a type `HttpListenerContext`.

For example,

```
var context = await listener.GetContextAsync();
```

The type of `context` is `HttpListenerContext`.

(Or more precisely, a nullable type `HttpListenerContext?` (because we use the implicit `var` declaration), but that is not very significant in this particular context since we, and the compiler, know that `context` is not, and cannot, be null since `GetContextAsync()` returns a non-null `HttpListenerContext` object.)

In this sample code, the `ProcessTextResponse()` method and the `ProcessJsonResponse` method write to the (HTTP) output stream.

Note that they are made `async` using a "trick", e.g., by calling an async method `await Task.Delay(1)`. Otherwise we could have simply created and returned a `Task` object.

In this example, for illustration, we used the async function delegate, `Func<HttpListenerContext, Task>`, although these two particular methods do not take too long to complete.

In general, in the context of Web programming, the handlers may need to do long processing or accessing databases, etc. They *may* take long time to finish, and it is a best practice to use async methods.

`JsonSerializer`

24.3. Pair Programming - Async Programming, `Func` *Delegates,* `HttpListener`*,*

Summary

We finally had a chance to use `async` and `await` programming style in this lesson. As stated, this is one of the most important features of the C# programming language. In any large project, async programming is indispensible.

Author's Note

"Polyglotting"

Hope you learned something new from the lessons in this part. If any of the topics we covered here was not clear to you, then you can always go back and repeat the lessons, or you can refer to different resources on the Web.

Many people learn foreign languages for fun. Even if you have no plans to travel to Japan, for instance, at least not in the near future, you can still learn the Japanese language just for the fun of it. A lot of people do. A lot of people speak many languages.

Likewise, many programmers learn and use many different programming languages. Often, for practical reasons. But, sometimes, just for fun.

All programming languages are different. They offer different perspectives. They have different strengths and weaknesses. They have different areas of primary uses. For example, Python is now becoming *the* language for AI/machine learning. If you are interested in low-level systems programming, then you will have to use C/C++ or Rust.

Learn a new programming language just for the fun of it. If you use Java, for instance, at your work, then learn Rust. Use Rust to build something fun.

The author has used over 20 different programming languages over the years, sometimes by necessity, sometimes just for fun.

Here's a list of other (upcoming) books by the author, if you are interested:

- The Art of Go - Basics: Introduction to Programming in Go
- The Art of Python - Basics: Introduction to Programming in Modern Python
- The Art of Typescript - Basics: Introduction to Programming in Typescript and Javascript
- The Art of Rust - Basics: Introduction to Programming in Rust
- The Art of C++ - Basics: Introduction to Programming in Modern C++

Yes, they all have more or less the same titles, except for the language.

They are part of the "Learn Programming (Languages) for Fun" series.

Part III: Having Fun

Just keep taking chances and having fun. -Garth Brooks

Chapter 25. Shuffle Play

25.1. Problem

The music "playlist" is one of the best ways to listen to music nowadays. All big music streaming services like Spotify and Apple Music support both public and private playlists.

Playlists are generally ordered (e.g., from top to bottom), and they can be played in the preset order. Alternatively, they can be also played in a random order, known as the "shuffle play".

Let’s suppose that we have a list of songs, or a playlist. Implement an algorithm that shuffles the songs in a random order.

25.2. Discussion

The most efficient way to shuffle a list of items, like a deck of cards, is using an algorithm known as the Fisher-Yates shuffle. Cf. Fisher-Yates Shuffle [https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle].

Let's start by defining a type for playlist. A playlist contains a list of songs. To shuffle a list, we will need to make a copy of the list since we do not want to change the original (ordered) list. The type `Playlist` that we use for playlists will therefore have to support some kind of "clone" or "copy" operations.

It will also have to support comparison operations, for sorting. We can do that in a number of different ways, including explicitly implementing the `IComparable<T>` interface, or inheriting from a collection type, as we did in the earlier example, Orgs and Employees.

> We used the inheritance in defining the `Organization` type. But that is not a very common practice, that is, inheriting directly from a collection type. In that example, one of the reasons for doing so was to illustrate the use of the collection initializer syntax for custom types, which was kind of "cool". ☺

In this lesson, we will use a more common technique known as "composition". In fact, composition is not that special. We simply include a variable of a type that we need to reuse/borrow as a field in our type definition. And, we delegate certain functionalities to the contained field variable.

For instance, conceptually a `Playlist` *is* a list of `Songs`. But, instead of directly inheriting from a list of Songs, in the code sample, we "embed" a variable of a type `List<Song>` in the `Playlist` type.

We expose some List-related APIs, such as `Add()` and `Size`, and we simply reuse the implementation of `List<Song>`. For example,

```
public void Add(Song song) => list.Add(song);
```

```
public int Size => list.Count;
```

In this case, we even "changed" the property name (e.g., from `Count` to `Size`), which is not possible through inheritance. (Readers should be familiar with the modern, rather terse, C# syntaxes by now. The first one is easy. It is an "expression-bodied" method. What about the second one? `Size` is a "readonly" property (no setter). The getter returns `list.Count`.)

The only "limitation" of the composition-based code reuse is that we can only use public APIs of the embedded type. (Or, internal if it is in the same assembly.) In inheritance, we can reuse the parent class's protected members as well.

For the shuffle method, we (indirectly) rely on the List's indexers. The method implements the Fisher-Yates algorithm.

25.3. Sample Code Snippets - *shuffle-playlist*

We create a type `Playlist` to encapsulate the data, a list of songs, and to provide an API, in particular, the "Shuffle()" method. Here's the test driver for `Playlist`.

shuffle-playlist/Program.cs (lines 5-15)

```
var playlist = new Playlist("My Playlist", new Song[] {
    new("title 1", "song 1"),
    new("title 2", "song 2"),
    new("title 3", "song 3"),
});
playlist.Add(new("title 4", "song 4"));
Console.WriteLine($"Original: {playlist}");
```

```

var shuffled = playlist.Shuffle();
Console.WriteLine($"Shuffled: {shuffled}");
Console.WriteLine($"Original: {playlist}");
```

shuffle-playlist/Program.cs (lines 20-27)

```
var list = new Playlist("New Playlist");

var t1 = Task.Run(() => AddSongs(list, "a"));
var t2 = Task.Run(() => AddSongs(list, "b"));
var t3 = Task.Run(() => ShuffleList(list));
await Task.WhenAll(t1, t2, t3);

Console.WriteLine($"Playlist = {list}");
```

shuffle-playlist/Program.cs (lines 29-49)

```
const int repeats = 100;

static async Task AddSongs(Playlist playlist, string suffix) {
    Random rng = new();

    for (var i = 0; i < repeats; i++) {
        var song = new Song($"t{i}{suffix}", $"s{i}");
        playlist.Add(song);
        await Task.Delay(rng.Next(0, 1000));
    }
}

static async Task ShuffleList(Playlist playlist) {
    Random rng = new();

    for (var i = 0; i < repeats; i++) {
        var shuffled = playlist.Shuffle();
        Console.WriteLine($"Shuffled = {shuffled}");
        await Task.Delay(rng.Next(0, 1000));
    }
}
```

The first part of the main program creates an instance of `Playlist`. And, it first tests the public method `Add()`. It calls the `shuffle()` method on the Playlist object, and then it prints out the shuffled and original Playlists to make sure that the original list has not changed after the shuffle.

The second part tests the multi-threaded behavior of a `Playlist` object. It runs three different threads which do addition and shuffling *concurrently*. The `AddSongs()` and `ShufflieList()` methods are defined as “async” methods. They in turn use a simple `async` method, `Task.Delay()`. (As stated, an `async` method can be called with `await` only in `async` methods.)

We define `Playlist` as a sealed class.

shuffle-playlist/Playlist.cs (lines 3-72)

```
public sealed class Playlist {
    private readonly object listLock = new();
    private static readonly Random rng = new();

    public string Name { get; }

    private readonly List<Song> list;

    public Playlist(string name) : this(name, new List<Song>()) {
}
    public Playlist(string name, Song[] songs) : this(name, new
List<Song>(songs)) { }
```

```
    public Playlist(string name, List<Song> list) => (Name, this.list) = (name, list);

    public Playlist Copy() => GetRange(0, Size);

    private Playlist GetRange(int index, int count) {
        lock (listLock) {
            return new(Name, list.GetRange(index, count));
        }
    }

    public Song[] ToArray() {
        lock (listLock) {
            return list.ToArray();
        }
    }

    public void Add(Song song) {
        lock (listLock) {
            list.Add(song);
        }
    }

    public Song this[int index] {
        get {
            lock (listLock) {
                return list[index];
            }
        }
        set {
            lock (listLock) {
                list[index] = value;
            }
        }
    }

    public int Size {
        get {
            lock (listLock) {
                return list.Count;
            }
        }
    }

    public Playlist Shuffle() {
        var copied = Copy();

        for (var i = 0; i < copied.Size - 1; i++) {
            var j = rng.Next(0, copied.Size - i) + i;
            (copied[i], copied[j]) = (copied[j], copied[i]);
        }

        return copied;
    }

    public override string ToString() {
        lock (listLock) {
            return $"{Name} => [{string.Join(", ", list.Select(s => s.ToString()))}]";
        }
    }
}
```

As indicated, the `Playlist` type contains a list of `Songs` as a private member field.

```
private readonly List<Song> list;
```

Note that we also mark it as `readonly`. As a general rule, you should always *try to* make your internal data readonly or immutable by default, unless it is required otherwise. In the modern multi-threaded, multi-processed runtime environments, dealing with mutable data is much more complicated. (Clearly, it is not always possible to make all data immutable, however.)

Declaring the variables of a collection type as `readonly` provides only limited protection. For example, even though the variable `list` is readonly and cannot be changed, one can easily change *its elements*.

In fact, this is true for all reference types. The language constructs like `readonly` only provides protection for the variable itself. But, for reference type variables, we mostly do not care about the reference values. We primarily care about the values that they reference. But, C# has no fundamental support at the language level to make the underlying data immutable that the references point to. (This is not only for C# but in general for all C-style languages.)

This is one of the reasons why the (immutable) values, and the value types, might be preferred over the references, and the reference types, in many situations.

We discussed the value type vs the reference types in earlier lessons. And, the "data type" vs everything else. The truth is, we barely scratched the surface. There is only so much that an (introductory) book like this can teach you. Ultimately, you will have to develop an "intuition", if you will, through experience.

But, as a general rule of thumb, related to this lesson, always strive to use immutable values. If you use a struct type, then make it "readonly", as well as all its fields, so that the values of the type are guaranteed to be immutable.

If you create a type using `record` (for "data"), then again make it immutable. All properties should be readonly, and you should not provide any method that can mutate the data.

> In practice, it is not always possible. Some types are sometimes more like data, and at other times, it is their "behaviors" that are more important. Some types have dual nature of values and references. Regardless, this dichotomy of thinking is very useful when designing a large software system.

In the sample code of this lesson, we use the C#'s `lock` statement to "protect" the data `list`. It is a common practice to create an (empty) object instance as a lock object.

```
private readonly object listLock = new();
```

The `listLock` object is a surrogate for the real resource that we are really protecting, namely, `list`. We could have just used the reference variable `list` as a lock, but using a proxy like this is not uncommon. In some cases, the values that we are protecting cannot be used as a lock.

In a multi-threaded environment, only one thread can hold any given lock. For instance,

```
public void Add(Song song) {
    lock (listLock) {
        list.Add(song);
    }
}
```

In order to be able to add a new item to the `list`, we need to hold the lock first, using the `lock (listLock) {}` statement. While the thread executing this method is holding the lock (inside the curly brace pair), no other thread can access `list`. None can modify the `list` or even read it without first acquiring the lock (because our API design does not allow it). Doing so could result in ending up using inconsistent data based on which thread accesses the data first. This is known as the "race condition" in computer science.

If you are new, or coming from languages like Javascript, then this may all sound like a gibberish. In some languages like Java, multi-threading is such an essential part of programming. In some other languages like Python, or even Go, it is less so (e.g., because they provide higher-level/easier-to-use APIs). Javascript runtimes are single-threaded. (But, "asynchronous programming" is still a very important part of modern Javascript.)

C# lies somewhere in between. In C#, threading APIs are considered "low-level" APIs. We mostly program using "high-level" constructs like `async`/`await`, as we discussed in the previous part.

But, the C# programmers still have to be aware of the implications of running the program in the multi-threaded runtime environments (like .NET or anywhere else).

The important thing to note is that the data field like `list` in an instance of a type (class, record, struct) is *shared* by multiple threads, and we need to coordinate their access (unless they are (truly) immutable). (As stated, declaring the reference variables `readonly` is not sufficient unless the type is designed to be "truly immutable".)

That is why we use the `lock` statement everywhere we access the variable in `Playlist`. The variables of the collection type `IList<T>` are not "thread-safe", as we say, and they need to be "protected".

We also declare the `rng` variable of a type `Random` readonly.

```
private static readonly Random rng = new();
```

The `Random` type appears to be pretty much immutable. Once you set the seed, via the constructor, there is no way to "write" to the variable of type `Random`. (The default constructor of `Random` uses the current time as a seed by default.) But, again `Random` is not an immutable type by its very nature. First of all, it is a reference type. And, it is not a "data" type.

`Random` prescribes a behavior. It generates a sequence of random numbers. Depending on how you access it, you get a different number. They are "deterministic" (in the sense that they are "pseudo-random" numbers). Granted. Even so, suppose that you have the multiple threads that request the random numbers from the same `object`. They all may end up getting different sequences of numbers unless they are synchronized in some way. (As one can easily guess, `System.Random` is implemented as a `class`.)

As stated, the `readonly` keyword provides limited protections for the reference type variables. But, it is still recommended to use `readonly`, when applicable, at least to protect the references themselves, if not the data they are referencing.

The `Shuffle()` method returns a copy of itself, with its internal Song list shuffled. This is to preserve the original ordering, as indicated earlier.

Note that the `Shuffle()` method itself does not use the `lock` statement. The `Copy()` method is synchronized. In this particular case, that should be enough. You can also use nested lock statements, in which a lock statement block includes another lock statements (either directly or via another method call). Only the thread that currently has the outer lock can acquire any inner lock.

Note that improperly implemented synchronization can lead to a situation known as a "dead lock". All, or some, threads cannot proceed because they cannot acquire a lock, which eventually leads to a situation in which there is no real progress in the program execution.

As stated, if you are not entirely familiar with multi-threaded programming, it is best to stick with C#'s async/await features.

A `Song` is defined as an immutable record type.

shuffle-playlist/Song.cs (lines 3-5)

```
public record Song(string Title, string Singer) {
    public override string ToString() => $"({Title}: {Singer})";
}
```

Although there is no explicit declaration, the record is implemented with a positional record syntax and it keeps no other internal data. It has no other data accessors or methods. (The properties, Title and Singer, are `readonly` by construction.) Hence, `Song` is truly immutable for all intents and purposes.

Clearly, the simplest immutable record type is the one declared with no body. For instance,

```
public record Song(string Title, string Singer);
```

In the particular example of this lesson, we just needed to override the `object.ToString()` method, which is (supposed to be) semantically readonly.

In C#, currently (as of versions 9.0 and 10.0), there is no way to explicitly declare or indicate that a particular record type is truly immutable (other than using the pure positional record type). It is up to the programmer to practice safe programming.

Not all types need to be created thread-safe. The overall program may need to run thread-safe, depending on its use cases, but not all individual types, and their instances, need to be thread-safe.

Chapter 26. Ring Buffer

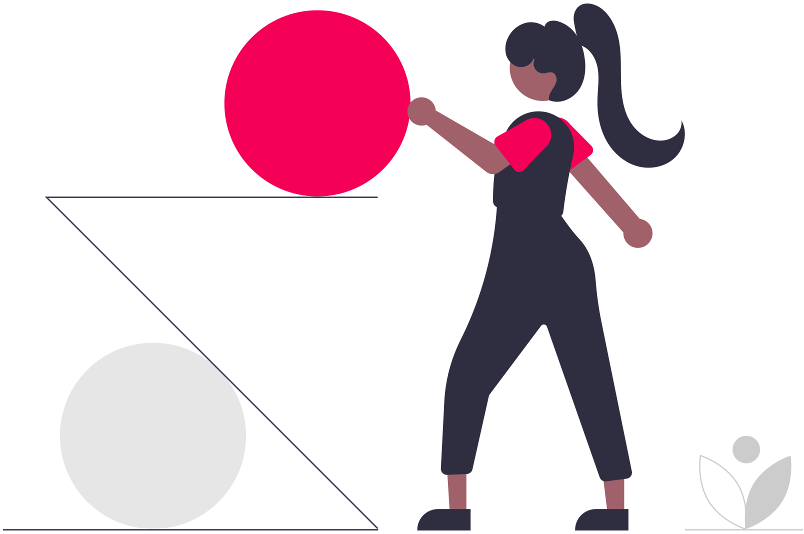
26.1. Problem

A ring buffer is a data structure that behaves like an array with a circle-like topology: Circular buffer [https://en.wikipedia.org/wiki/Circular_buffer]. It has many uses. It can be used as an efficient fixed size FIFO queue (first in first out). It can also be used as an MRU list (most recently used), among other things.

Let's implement a circular data structure that supports both the FIFO queue and MRU list use cases.

In a typical FIFO queue use case, if the finite-size queue becomes "full", then we will need to throw an error or otherwise we should be able to indicate this situation in some way (e.g., to the client program) so that no data is lost. When a finite-size buffer is used for temporarily caching data, however, as in the typical use case of an MRU list (e.g., in the "recently open file list" menu in many file-oriented apps), we can just overwrite the old/oldest data with new data.

We will implement a ring buffer data structure as a library and create two test apps to use this data structure, one for a FIFO queue and the other for an MRU list.



26.2. Discussion

Throughout this book, we mostly use a single project sample code. This lesson is an exception. We create four projects. One library, and its test project, and two app projects.

There is some overhead. Even for simple unit testing, we will need to create a separate project. This is really unfortunate but that is the way it is, currently, in the Microsoft .NET ecosystem.

We can create a library assembly project using the `dotnet` CLI:

```
dotnet new classlib
```

You can achieve the same if you use Visual Studio or other IDEs.

As far as the project XML file is concerned, the real difference is the `OutputType` property. Instead of `exe`, we use `library` for a library project:

```
<OutputType>library</OutputType>
```

You can create an "XUnit" test project as follows:

```
dotnet new xunit
```

It is customary to put app and library projects under a folder named `src`, and test projects under a folder named `test`. But that is not required.

You can create two app projects using `dotnet` as well (in appropriate folders):

```
dotnet new console
```

As indicated before, however, the `dotnet new` scaffolding is pretty minimal (as far as the console apps are concerned), and you can just copy the existing project file and any C# files, and make necessary modifications, as a starting point for your new project(s).

You can manage multiple projects using a "solution" file (which Visual Studio uses by default). But, again, that is not required. The more important thing is to add references to the dependent projects. For instance, in the example projects of this lesson, the library test project depends on the "RingBuffer" library project, and both app projects also depend on this library project.

We can use the `dotnet add reference` to add project dependencies. If you use the dotnet CLI, the `--help` option comes in handy. For instance,

```
$ dotnet add reference --help

Usage: dotnet add <PROJECT> reference [options] <PROJECT_PATH>
```

```
Arguments:
  <PROJECT>        The project file to operate on. If a file is not
specified, the command will search the current directory for one.
  <PROJECT_PATH>   The paths to the projects to add as references.

Options:
  -h, --help                        Show command line help.
  -f, --framework <FRAMEWORK>   Add the reference only when targeting
a specific framework.
  --interactive                     Allows the command to stop and wait
for user input or action (for example to complete authentication).
```

For example, we can do the following to add a reference to the library project in the test project.

```
dotnet add reference ../RingBuffer/ring-buffer.csproj
```

Or, we can manually update the project files. For example, the following is taken from the test project.

```
<ItemGroup>
  <ProjectReference Include="..\..\src\RingBuffer\ring-buffer.csproj"
/>
</ItemGroup>
```

For this sample project, the overall directory structure looks like this:

```
book/03-having-fun/code/ring-buffer$ tree

.
```

```
├── src
│   ├── MRU
│   │   ├── bin
│   │   ├── List.cs
│   │   ├── obj
│   │   ├── Program.cs
│   │   └── ring-buffer-mru.csproj
│   ├── Queue
│   │   ├── bin
│   │   ├── obj
│   │   ├── Program.cs
│   │   ├── Queue.cs
│   │   └── ring-buffer-queue.csproj
│   └── RingBuffer
│       ├── bin
│       ├── Buffer.cs
│       ├── obj
│       └── ring-buffer.csproj
└── test
    └── RingBufferTests
        ├── bin
        ├── BufferTest.cs
        ├── obj
        └── ring-buffer-tests.csproj

60 directories, 231 files
```

26.3. Sample Code Snippets - *ring-buffer*

There can be many different ways to implement a circular buffer. Here’s one implementation:

ring-buffer/src/RingBuffer/Buffer.cs (lines 4-72)

```
public sealed class Buffer<T> : IEnumerable<T> {
    private readonly int size;
    private readonly T[] ring;

    // head >= tail ????
    private int head = 0; // "Next" index
    private int tail = 0;

    public bool Overwrite { get; set; }

    public Buffer(uint capacity, bool overwrite = false) {
        ring = new T[capacity + 1];
        size = ring.Length;
        Overwrite = overwrite;
    }

    public bool Empty => head == tail;
    public bool Full => head == (tail + size - 1) % size;
    public int Capacity => size - 1;
    public int Length => (head == tail) ? 0 : (head - tail + size) % size;

    // Translate index into "ring index".
    // The "ring index" runs from head to tail, backward.
    public T this[int index] {
        get {
            index = (index % Capacity + Capacity) % Capacity;
            return ring[(head - index - 1 + size) % size];
        }
        set {
            index = (index % Capacity + Capacity) % Capacity;
            ring[(head - index - 1 + size) % size] = value;
        }
    }

    public void PushFront(T item) {
        if (Overwrite) {
            ring[head++] = item;
            head %= size;
            if (Full) {
                tail = (tail + 1) % size;
            }
        } else {
            if (Full) {
                throw new OverflowException("Buffer is full");
            }
            ring[head++] = item;
            head %= size;
        }
    }

    public T PopBack() {
        if (Empty) {
            throw new InvalidOperationException("No items in Buffer");
        }
        var item = ring[tail++];
        tail %= size;
        return item;
    }

    public IEnumerator<T> GetEnumerator() {
        for (var i = 0; i < Length; i++) {
            yield return this[i];
        }
    }

    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
```

We define our RingBuffer as a generic sealed class Buffer<T> which implements the IEnumerable<T> interface. There is no further constraint on type T. It can be used with any type, from object to any of its derived class/record or struct types.

Earlier in the book, we mentioned the importance of implementing System.IEnumerable<T> when the type is to be used in foreach as a collection. In fact, its explicit declaration is not needed. As long as the type provides an implementation of IEnumerator<T> GetEnumerator() method, it can be used in the foreach statement. (Note that this feature is new as of C# 9.0.)

This comes in handy when we need to use a variable of a collection type that is not under our control. In that case, we can add an extension method to the type that has the same signature as IEnumerator<T> System.IEnumerable<T>.GetEnumerator().

We will leave it to the readers to go through the implementation of Buffer. It is a good exercise.

But one thing to note is that the access of the ring field of Buffer is not synchronized. ring is of a type T[], an array of elements of type T. An array is a reference type. The use of Buffer is in general not thread safe regardless of the element type T. As indicated, however, not every type has to be thread-safe. It is the overall program that matters.

Here’s a simple test file for the Buffer<T> class:

ring-buffer/test/RingBufferTests/BufferTest.cs (lines 5-13)

```
public sealed class BufferTest {
    [Fact]
    public void TestBasics() {
        var cap = 3u;
        var buffer = new Buffer<int>(cap);

         Assert.Equal(0, buffer.Length);
         Assert.Equal((int)cap, buffer.Capacity);
```

The special syntax `[Fact]` is called an "attribute". In this particular case, the attribute `[Fact]` is defined in the `Xunit` namespace, and it declares the method, following the attribute, as a test method.

You can run all tests in the folder, `RingBufferTests`, as follows (from the "root folder"):

```
dotnet test test/RingBufferTests
```

Or, if you have a solution file, then it is a little bit more convenient. You can just run `dotnet test` from the root folder (e.g., the parent of both `src` and `test`) or from the test project folder (even without a solution file).

Here's a finite-size FIFO queue implementation using `Buffer<T>`.

ring-buffer/src/Queue/Queue.cs (lines 4-35)

```
public class ShortQueue<T> : IEnumerable<T> {
    private readonly object bufferLock = new();

    private readonly Ring.Buffer<T> buffer;
    public uint Capacity { get; init; }
    public int Count => buffer.Length;

    public ShortQueue(uint capacity) {
        Capacity = capacity;
        buffer = new Ring.Buffer<T>(Capacity);
    }

    public void Enqueue(T item) {
        lock (bufferLock) {
            buffer.PushFront(item);
        }
    }

    public T Dequeue() {
        lock (bufferLock) {
            return buffer.PopBack();
        }
    }

    public IEnumerator<T> GetEnumerator() => buffer.GetEnumerator();

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

And, a test driver main program:

ring-buffer/src/Queue/Program.cs (lines 3-31)

```
var cap = 2u;
var queue = new ShortQueue<int>(cap);

Console.WriteLine($"len = {queue.Count}");

var e1 = 5;
queue.Enqueue(e1);
var e2 = 7;
queue.Enqueue(e2);
```

```

Console.WriteLine($"len = {queue.Count}");
lock (queue) {
    foreach (var item in queue) {
        Console.WriteLine($"item = {item}");
    }
}

var e3 = 9;
try {
    queue.Enqueue(e3);
    Console.WriteLine($"len = {queue.Count}");
    lock (queue) {
        foreach (var item in queue) {
            Console.WriteLine($"item = {item}");
        }
    }
} catch (Exception ex) {
    Console.WriteLine(ex);
}
```

Note that we use a composition. `ShortQueue<T>` includes `Buffer<T>` as a member field, just like `Buffer<T>` includes `T[]` as a member field.

As stated, using composition for code reuse is much more common than using inheritance. Inheritance has its place, but as a general rule, you should prefer composition over inheritance unless there is an explicit reason to use inheritance.

Note that we synchronize the use of the internal field, `buffer`. Since an object of Type `Buffer` is not thread-safe, we should synchronize its uses in some way. And, the `lock` statement is one of the simplest and the most common ways to protect the shared resources.

If we create a non-thread-safe type, for example, for

> `ShortQueue<T>`, then again we should protect its uses when we use the variables of the type.

One more thing to note is that there is really no easy way to synchronize the implementation of `GetEnumerator()` method.

The method returns a (reference) variable of type `IEnumerator<T>`. Once the `GetEnumerator()` method returns an `IEnumerator<T>` object, the caller can do whatever it wants to do with the "data" (i.e., the embedded buffer items), which are in turn the elements of the array type `T[]`.

Hence, `ShortQueue<T>` is not fully thread-safe, and its user should synchronize the data uses in some way.

That is what the Queue test program does in this example. Note the use of `lock(queue)`. It directly uses the resource that we are protecting as a lock object. As stated, it is also a common pattern to do so when that is possible. The values of value types, and strings (whose values are "interned"), and any variables of the types that have copy semantics, may not be suitable as a lock object.

In this particular example, the resource to synchronize its use is a variable of a reference type, `Ring.Buffer<T>`, whose resource needs synchronization, is in turn a variable of the reference type `T[]`. Whether `T` is a value type or a string or another reference type has no bearing in this example.

Another use case of `Buffer<T>` is an MRU list.

ring-buffer/src/MRU/List.cs (lines 4-42)

```
public sealed class ShortList<T> : IEnumerable<T> {
    private readonly object bufferLock = new();

    private readonly Ring.Buffer<T> buffer;
    public uint Capacity { get; init; }
    public int Count => buffer.Length;

    public ShortList(uint capacity) {
        Capacity = capacity;
        buffer = new Ring.Buffer<T>(Capacity, overwrite: true);
    }

    public T this[int index] {
        get {
            lock (bufferLock) {
                return buffer[index];
            }
        }
        set {
            lock (bufferLock) {
                buffer[index] = value;
            }
        }
    }

    public void Add(T item) {
        lock (bufferLock) {
            buffer.PushFront(item);
        }
    }

    public IEnumerator<T> GetEnumerator() =>
buffer.GetEnumerator();

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

And, its test driver:

ring-buffer/src/MRU/Program.cs (lines 3-21)

```
var cap = 3u;
var list = new ShortList<int>(cap);

list.Add(1);
list.Add(2);
Console.WriteLine($"len = {list.Count}");
lock (list) {
    foreach (var item in list) {
        Console.WriteLine($"item = {item}");
    }
}

list.Add(3);
Console.WriteLine($"len = {list.Count}");
lock (list) {
    foreach (var item in list) {
        Console.WriteLine($"item = {item}");
    }
}
```

We will leave it to the readers to understand this code sample, and how it differs from the use case of the fixed size FIFO queue.

We could have created additional test projects for testing each of these apps (e.g., using NUnit or XUnit or other C#/.NET testing frameworks). But, for illustration purposes, the "main programs" should suffice. All sample programs in this book are written like they are "real programs", which we can potentially "publish", if desired.

Chapter 27. LED Digital Clock

27.1. Problem

Digital desktop clocks are not as common as before. One can still find them in places like hotel rooms. Most digital watches use LCD screens, and again those using LED-style digital displays are relatively hard to find.

Let’s suppose that we need to program a digital clock, which consists of four digits, and each digit consists of 7 LEDs.

Something like this, using the vertical bar and underscore characters as the "light emitting diodes":

```
 _
|_|
|_|
```

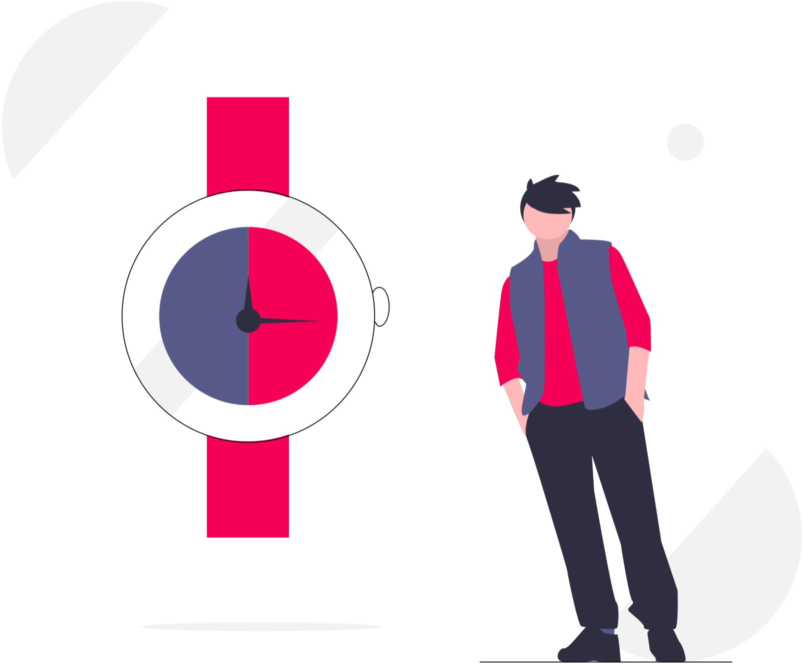
Here’s a representation of all ten digits.

```
 _       _   _       _   _   _   _   _
| |   |  _|  _| |_| |_  |_    | |_| |_|
|_|   | |_   _|   |  _| |_|   | |_|   |
```

We are showing only the LEDs that are "turned on", in this notation.

The problem of this lesson is to create a program that "controls" the digital clock display. Given a time, we need to turn on and off particular LEDs to indicate the time on the display.

To be concrete, we have 28 LEDs (4 times 7), and the goal is to control these 28 LEDs to display a particular time.



27.2. Discussion

We will use a "dummy display" on the terminal. But, in theory, we can use the program to control the real digital clock display, depending on what kind "interface" the display exposes as "API".

Here’s an example time display:

```
$ dotnet run

Current time is:
 _     _  _
 _|  ||_ |_
|_   | _| _|

Current time is:

 _     _  _
 _|  ||_ |_
|_   | _||_|

Current time is:

 _     _  _
 _|  ||_   |
|_   | _|  |
^C
```

The sample app displays a new time every 60 seconds. We killed the program using the interrupt signal (Control+C on Unix, for instance) in this example.

27.3. Sample Code Snippets - *led-clock-digital*

The main program merely calls the async `Run()` method of an `LED.Clock` object, which in turn calls the `DisplayCurrentTime()` method every 60 seconds by default.

led-clock-digital/Program.cs (lines 3-4)

```
var clock = new Clock();
await clock.Run();
```

As indicated before, a different `Main()` method is generated from the top-level statements based on a few conditions: Whether they use the command line arguments or not, whether they return a value at the end, and whether they call any `async` methods with `await`.

The valid signatures of the C# `Main` method are the following:

```
public static void Main();
public static int Main();
public static void Main(string[] args);
public static int Main(string[] args);
public static async Task Main();
public static async Task<int> Main();
public static async Task Main(string[] args);
public static async Task<int> Main(string[] args);
```

Note that 2 times 2 times 2 is 8. One can use any of these 8 as the main method of a program (e.g., in a class or interface). In the case of the top-level statements, as stated, the compiler generates the appropriate `Main()` method.

`Clock` is implemented as a `sealed class`, not as a struct or record, since we are interested in its behavior, e.g., the "clock-ness", if you will.

led-clock-digital/Clock.cs (lines 4-26)

```
public sealed class Clock {
    const int NumDigits = 4;

    private readonly ILEDDisplay display;

    public Clock() => display = new LEDDisplay(NumDigits);

    public async Task Run(int milliDelay = 60_000) {
        while (true) {
            Console.WriteLine("Current time is:");
            DisplayCurrentTime();
            await Task.Delay(milliDelay);
            Console.WriteLine();
        }
    }

    public void DisplayCurrentTime() {
        var time = DateTime.Now;
        var str = $"{time.Hour:D2}{time.Minute:D2}";
        var digits = str.Select(c => Digit.Get(c)).ToArray();
        display.Show(digits);
    }
}
```

The `Clock` class encapsulates the basic characteristics of a "clock". It gets the time, from the simple .NET API `DateTime.Now` in this case, and it displays the time (on a regular interval). That *is* a clock.

Note that the implementation of the `DisplayCurrentTime()` method uses another LINQ method, as defined as an extension method on `IEnumerable<T>`. The `Select()` method takes an argument of a delegate type, `Func<char, Digit>` in this particular case.

The method is called with a lambda function, `c => Digit.Get(c)`, to convert a char to the corresponding "Digit". It returns a value of `IEnumerable<Digit>`.

Our hypothetical display device uses `Digits` to communicate the information regarding which LEDs should be turned on or off.

led-clock-digital/Common/Digits.cs (lines 3-10)

```
public sealed record Digit {
    public static readonly Digit None = new(new byte[] {
            0, 0, 0, 0, 0, 0, 0
        });

    public static readonly Digit Zero = new(new byte[] {
            1, 1, 1, 0, 1, 1, 1
        });
```

led-clock-digital/Common/Digits.cs (lines 48-67)

```
    public static Digit Get(char c) =>
        c switch {
            '0' => Zero,
            '1' => One,
            '2' => Two,
            '3' => Three,
            '4' => Four,
            '5' => Five,
            '6' => Six,
            '7' => Seven,
            '8' => Eight,
            '9' => Nine,
            _ => None,
        };

    private readonly byte[] leds;

    private Digit(byte[] leds) {
        this.leds = leds;
    }
```

(Note that we are not including the entire program here. For a full code sample, refer to the appendix.)

A type `Digit` is essentially a simple wrapper around the type `byte[]` (for 7 LEDs), again using the composition, and it defines a single public API, the `Get()` method.

The `Get()` method uses the switch expression to convert a char to one of the ten predefined Digits. For a char that is not a digit/number, we simply return the "default" value for the Digit type, `None`.

There are a couple of things to note about the `Digit` type. First, its (only) constructor is `private`. Nobody can instantiate a Digit. They can only use the predefined digits plus `None`.

Another thing to note is that `Digit` is a `record` type. Not a `struct` type. In general, it is best for a value type not to include any reference type fields. It is generally hard to enforce value semantics on a value that includes any reference variables. In this particular case, however, since there is no way to update internal field, `leds`, since the constructor is private and there are no setters, we could have also used `struct`.

Note the implementation of the constructor. We simply copy the reference of the argument `leds`. If anybody else has an access to the reference variable, they can easily change the values of the elements of `leds`, which is not the behavior we desire.

In general, a better implementation would have been to "copy", or "clone", the argument array and keep the "private copy" internally. As stated, software design is about a tradeoff. Doing so would have been more expensive in terms of the computational requirements.

> The 7 LEDs are arranged in a particular order. Again, a code comment would have been nice in situations like this. Can you guess the mapping between the array indices, `0` ~ `6`, and the particular LEDs?

We use the interface, `ILEDDisplay`, for "abstraction". That is, our `Clock` implementation uses the interface and we can "plug" in any `ILEDDisplay` implementations later (possibly, with some minor changes).

In this case, for instance, we may ultimately want to use our `Clock` to interface with real digital LED clocks.

led-clock-digital/Common/Display.cs (lines 3-9)

```
public interface IDisplay {
    void Clear();
}

public interface ILEDDisplay : IDisplay {
    void Show(Digit[] digits);
}
```

In software design, "abstraction" is a very powerful strategy, with broad applicability. Using interfaces to "abstract away" the implementation details is one of the most common "abstraction" strategies.

For concreteness, in this lesson, we use a "fake" ASCII display. The types are defined in the `LED.Ascii` namespace. The `Number` record is immutable, just like the `Digit` type.

led-clock-digital/Ascii/Number.cs (lines 4-14)

```
sealed record Number {
    internal static readonly Number None = new(new char[,] {
                { ' ', ' ', ' ' },
                { ' ', ' ', ' ' },
                { ' ', ' ', ' ' },
            });
    internal static readonly Number Zero = new(new char[,] {
                { ' ', '_', ' ' },
                { '|', ' ', '|' },
                { '|', '_', '|' },
            });
```

led-clock-digital/Ascii/Number.cs (lines 61-104)

```
    private static readonly Dictionary<Digit, Number> digits =
new() {
        { Digit.Zero, Zero },
        { Digit.One, One },
        { Digit.Two, Two },
        { Digit.Three, Three },
        { Digit.Four, Four },
        { Digit.Five, Five },
        { Digit.Six, Six },
        { Digit.Seven, Seven },
        { Digit.Eight, Eight },
        { Digit.Nine, Nine },
    };

    internal static Number Convert(Digit d) =>
        digits.ContainsKey(d) ? digits[d] : None;

    private readonly char[,] leds;

    private Number(char[,] leds) {
        this.leds = leds;
    }

    internal int Rows => leds.GetLength(0);
    internal int Cols => leds.GetLength(1);

    internal char Get(int i, int j) {
        if (i < 0 || i >= Rows || j < 0 || j >= Cols) {
            throw new ArgumentOutOfRangeException($"Invalid index:
({i},{j})");
```

```
        }
        return leds[i, j];
    }

    public override string ToString() {
        var sb = new StringBuilder();
        for (var i = 0; i < Rows; i++) {
            for (var j = 0; j < Cols; j++) {
                sb.Append(leds[i, j]);
            }
            if (i < Rows - 1) {
                sb.Append('\n');
            }
        }
        return sb.ToString();
    }
```

On the other hand, we also use a mutable type, `Cell`, which represents a sort of a display panel for one digit. Clearly, the display on the `Cell` will change depending on the actual number to be displayed, at any given moment.

led-clock-digital/Ascii/Cell.cs (lines 3-34)

```
sealed class Cell {
    private readonly char[,] leds;

    internal int RowCount => leds.GetLength(0);
    internal int ColCount => leds.GetLength(1);

    internal Cell() : this(Number.None) { }
    internal Cell(Number number) {
        leds = new char[number.Rows, number.Cols];
        Set(number);
    }

    internal char Get(int i, int j) {
        return leds[i, j];
    }

    internal void Set(Number number) {
        for (var i = 0; i < RowCount; i++) {
            for (var j = 0; j < ColCount; j++) {
                leds[i, j] = number.Get(i, j);
            }
        }
    }

    internal void Reset() {
        for (var i = 0; i < RowCount; i++) {
            for (var j = 0; j < ColCount; j++) {
                leds[i, j] = ' ';
            }
        }
    }
}
```

The `Cells` static class includes an extension method, `ToEnumerators()`, to support a "two dimensional" looping, which in turn uses another extension method, `ToEnumerator(i)`, for each row `i`.

led-clock-digital/Ascii/Cells.cs (lines 3-20)

```
static class Cells {
    internal static IEnumerable<IEnumerable<char>>
ToEnumerators(this List<Cell> cells) {
        var r = cells[0].RowCount;
        for (var i = 0; i < r; i++) {
            yield return cells.ToEnumerator(i);
```

```
 8         }
 9     }
10 
11     private static IEnumerable<char> ToEnumerator(this List<Cell> cells, int i) {
12         var len = cells.Count;
13         var c = cells[0].ColCount;
14 
15         for (var j = 0; j < len * c; j++) {
16             var (x, y) = (j / c, j % c);
17             yield return cells[x].Get(i, y);
18         }
19     }
20 }
```

The `LED.Ascii.Display` class includes the concrete implementation for displaying the digits on the terminal.

led-clock-digital/Ascii/Display.cs (lines 3-5)

```
3 public abstract class Display : IDisplay {
4     public abstract void Clear();
5 }
```

led-clock-digital/Ascii/Display.cs (lines 7-41)

```
 7 public sealed class LEDDisplay : Display, ILEDDisplay {
 8     private readonly List<Cell> cells;
 9 
10     public int Size => cells.Count;
11 
12     public LEDDisplay(int size) =>
13         cells = Enumerable.Range(0, size).Select(i => new
Cell(Number.None)).ToList() ?? new();

    public override void Clear() {
        cells.ForEach(c => c.Reset());
        Show();
    }

    private void Show() {
        foreach (var m in cells.ToEnumerators()) {
            foreach (var n in m) {
                Console.Write(n);
            }
            Console.WriteLine();
        }
    }

    public void Show(Digit[] digits) {
        var arr = digits.Select(d => Number.Convert(d)).ToList();
        for (var i = 0; i < Size; i++) {
            if (i < arr.Count) {
                cells[i].Set(arr[i]);
            } else {
                cells[i].Reset();
            }
        }

        Show();
    }
}
```

One thing to note in this example is that we are using "nested namespaces". The sample code, in the folder `Ascii`, uses the following file-scoped namespace declaration:

```
namespace LED.Ascii;
// ...
```

This namespace `LED.Ascii` is "contained" in the namespace `LED`. In C#, you can have as many nested namespaces. This is equivalent to

```
namespace LED.Ascii {
    // ...
}
```

which is in turn equivalent (according to the C# formal grammar) to

```
namespace LED {
    namespace Ascii {
        // ...
    }
}
```

By using the file-scoped namespace (available as of C# 10.0), we are "saving" the precious space on the left-hand side. (Note the indentations of the main part of the code (`...`) in each namespace declaration.)

Chapter 28. Rubik’s Cube

28.1. Problem

The Rubik’s Cube was wildly popular when it first came out in the 1970s. It is still very popular among the children, and among many enthusiasts and "speedcubers". Cf. Rubik’s Cube [https://en.wikipedia.org/wiki/Rubik%27s_Cube].

This is a very interesting puzzle. Not very "complex", but it is still very difficult to visualize in mind. The space of all possible configurations of the Rubik’s Cube has a topology of 3 dimensional torus [https://en.wikipedia.org/wiki/Torus], as we will see shortly.

Now the question is, how to "represent" the configurations of the Rubik’s cube in a program? In particular, in this lesson, we will tackle a problem of "shuffling" a cube or randomizing its configurations.

Each side of the Rubik’s cube can be rotated by 90, 180, and 270 degrees. 180 and 270 degree rotations can be viewed as two and three 90 degree rotations, respectively. It is customary among the Rubik’s cube players, however, that counting 90 degree and 270 (-90) degree rotations as two distinct moves. A 180 degree rotation can still be viewed as two moves of one of these rotations.

The Rubik’s Cube has six sides or "faces" (it’s a cube after all), and hence, from any given configuration, we can make one of the 12 moves: A 90 degree clockwise or counterclockwise rotation for one of the 6 faces.

Each of the 6 faces is associated with a color, White (`W`), Blue (`B`), Red (`R`), Green (`G`), Orange (`O`), and Yellow (`Y`). The white and yellow faces are on the opposite sides of the cube. The blue and green faces are on the opposite sides. The red and orange faces are on the opposite sides.

It’s hard to represent a three-dimensional object on a two dimensional surface, but the Rubik’s cube can be drawn as follows:

```
  G
R W/Y O
  B
```

If we place the cube on the surface, and if we look down on it from the top, then we see the white face, and each of the four sides is G, R, B, and O. The yellow face is at the bottom.

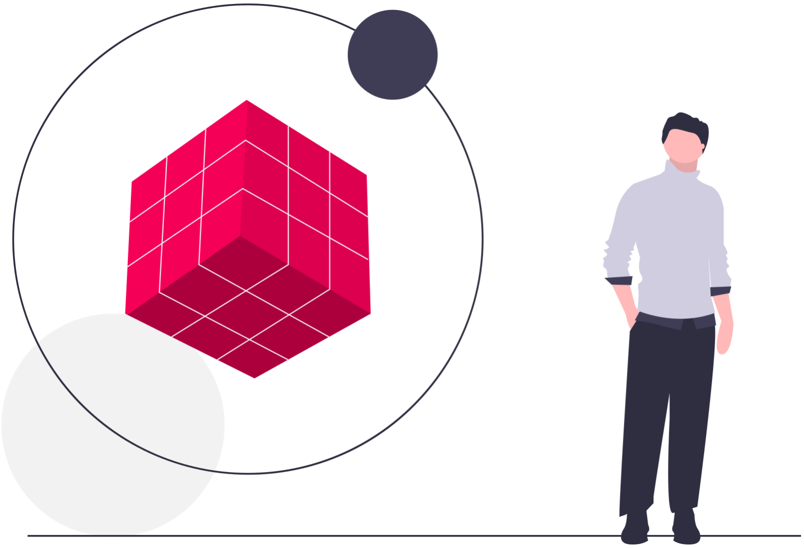
Each of the small piece on the cube faces is called a "tile". There are 54 tiles, 6 times 9 (3 by 3). There are 6 center tiles, 12 edge tiles, and 8 corner tiles.

```
54 = 1x6 + 2x12 + 3x8
```

Note that not all tiles are independent. There are, in fact, only 26 independent pieces in the Rubik’s cube. 6 center pieces are not movable. 12 edge pieces consists of two tiles and hence two colors. 8 corner pieces consists of three tiles of three different colors.

It might be useful to examine one of those cubes at this point if you happen to have one.

"Reading" a program is an order of magnitude more difficult than "writing". The sample code in this lesson may be difficult for you to read and understand. You can skip this lesson if you feel like it is too hard to read. If you would like to tackle the final final project, Rubik’s Cube Challenge, however, then the information from this lesson may turn out to be rather useful.



28.2. Discussion

A Rubik's cube is a "real" object. An object in the real world. It has a state (e.g., what tiles have what colors). It has some allowed operations, like a rotation of a face.

This is an ideal (real-word) object to represent as an "object" (in the sense of object oriented programming). The Rubik's cube has a state (e.g., the colors of the 54 tiles, as stated). And, the only way to change its state is through a well-defined set of interfaces (e.g., rotations of the faces by 90 degrees clockwise or counterclockwise, as stated).

All objects of OOP have states and interfaces. The challenge, for us developers, is to find a "good" representation of a real world object in a program. (The "good" may mean different things depending on the requirements.)

The readers are encouraged to think about this problem before reading the sample code. First of all, what are we trying to do? Then, what would be the best way (or, a "good enough" way) to achieve our goal? Is the OOP going to help? If so, how? And so forth.

(Knowing inheritance/polymorphism and all the technical details of the OOP will mean very little unless you understand the fundamental concepts of object oriented programming.)

28.3. Sample Code Snippets - *rubiks-cube*

The main program tests a few operations on the Rubik's cube.

rubiks-cube/Program.cs (lines 5-6)

```
5 var c = new Cube();
6 Console.WriteLine($"c =\n{c}");
```

rubiks-cube/Program.cs (lines 11-39)

```
11 c.RotateWhite();
12 Console.WriteLine($"c =\n{c}");
13 c.RotateWhite(false);
14 Console.WriteLine($"c =\n{c}");
15
16 c.RotateBlue();
17 Console.WriteLine($"c =\n{c}");
18 c.RotateBlue(false);
19 Console.WriteLine($"c =\n{c}");
20
21 c.RotateRed();
22 Console.WriteLine($"c =\n{c}");
23 c.RotateRed(false);
24 Console.WriteLine($"c =\n{c}");
25
26 c.RotateGreen();
```

```
Console.WriteLine($"c =\n{c}");
c.RotateGreen(false);
Console.WriteLine($"c =\n{c}");

c.RotateOrange();
Console.WriteLine($"c =\n{c}");
c.RotateOrange(false);
Console.WriteLine($"c =\n{c}");

c.RotateYellow();
Console.WriteLine($"c =\n{c}");
c.RotateYellow(false);
Console.WriteLine($"c =\n{c}");
```

rubiks-cube/Program.cs (lines 44-45)

```
c.Randomize();
Console.WriteLine($"c =\n{c}");
```

A `Cube` is a large class, probably, the biggest among all the sample code in this book.

rubiks-cube/Cube.cs (lines 5-42)

```
public sealed class Cube {
    private readonly Face[] faces = new Face[] {
            new(Color.W),
            new(Color.B),
            new(Color.R),
            new(Color.G),
            new(Color.O),
            new(Color.Y),
        };

    private Face W => faces[0];
    private Face B => faces[1];
    private Face R => faces[2];
    private Face G => faces[3];
    private Face O => faces[4];
    private Face Y => faces[5];

    private Random rand;

    public Cube() => rand = new Random();

    public void Randomize() {
        const int rep = 20;
        foreach (var u in Range(0, rep)) {
            var r = rand.Next(0, 12);

            var cw = r % 2 == 0;
            _ = (Color)(r / 2 + 1) switch {
                Color.W => RotateWhite(cw),
                Color.B => RotateBlue(cw),
                Color.R => RotateRed(cw),
                Color.G => RotateGreen(cw),
                Color.O => RotateOrange(cw),
                Color.Y => RotateYellow(cw),
                _ => false,
            };
        }
    }
```

rubiks-cube/Cube.cs (lines 47-62)

```
    private Triplet BsTopRowForW {
        get => B.TopRow;
        set => B.TopRow = value;
    }
    private Triplet RsTopRowForW {
        get => R.TopRow;
        set => R.TopRow = value;
    }
    private Triplet GsTopRowForW {
        get => G.TopRow;
        set => G.TopRow = value;
    }
    private Triplet OsTopRowForW {
        get => O.TopRow;
        set => O.TopRow = value;
    }
```

rubiks-cube/Cube.cs (lines 149-163)

```
    public bool RotateWhite(bool cw = true) {
        var row = BsTopRowForW;
        if (cw) {
            BsTopRowForW = OsTopRowForW;
            OsTopRowForW = GsTopRowForW;
            GsTopRowForW = RsTopRowForW;
            RsTopRowForW = row;
        } else {
            BsTopRowForW = RsTopRowForW;
            RsTopRowForW = GsTopRowForW;
            GsTopRowForW = OsTopRowForW;
            OsTopRowForW = row;
        }
        return true;
    }
```

rubiks-cube/Cube.cs (lines 245-257)

```
    public override string ToString() {
        var range = new List<int>() { 0, 1, 2 };
        var sb = new StringBuilder();
        sb.AppendLine(string.Concat(Repeat(new string('-', 7) + " ", 6)));
        foreach (var i in range) {
            for (var f = 0; f < 6; f++) {
                sb.Append($"[{string.Join(' ', range.Select(j => faces[f].GetRow(i)[j]).ToArray())}] ");
            }
            sb.AppendLine();
        }
        sb.Append(string.Concat(Repeat(new string('-', 7) + " ", 6)));
        return sb.ToString();
    }
```

But, there are a fair amount of repetitions, in fact, 6 times across the 6 different faces.

Here are some convenience types:

rubiks-cube/Face.cs (lines 4-50)

```
public sealed class Face {
    public Color Center { get; init; }

    private readonly Tile[,] tiles;

    private static readonly List<int> range = new() { 0, 1, 2 };

    public Triplet TopRow {
        get => range.Select(j => tiles[0, j]).ToArray();
        set => range.ForEach(j => tiles[0, j] = value[j]);
    }
    public Triplet RightCol {
        get => range.Select(i => tiles[i, 2]).ToArray();
        set => range.ForEach(i => tiles[i, 2] = value[i]);
    }
    public Triplet BottomRow {
        get => range.Select(j => tiles[2, 2 - j]).ToArray();
        set => range.ForEach(j => tiles[2, 2 - j] = value[j]);
    }
    public Triplet LeftCol {
        get => range.Select(i => tiles[2 - i, 0]).ToArray();
        set => range.ForEach(i => tiles[2 - i, 0] = value[i]);
    }

    internal Tile[] GetRow(int rowNum) => range.Select(j =>
tiles[rowNum, j]).ToArray();

    internal Face(Color center) {
        Center = center;

        var t = new Tile(Center);
        tiles = new Tile[3, 3]{
                {t, t, t},
                {t, t, t},
                {t, t, t},
            };
    }

    public override string ToString() {
        var sb = new StringBuilder();
        sb.AppendLine(new string('-', 7));
        foreach (var i in range) {
            sb.AppendLine($"[{string.Join(' ', range.Select(j =>
tiles[i, j]).ToArray())}]");
        }
        sb.Append(new string('-', 7));
        return sb.ToString();
    }
}
```

rubiks-cube/Triplet.cs (lines 3-24)

```
public sealed class Triplet {
    private Tile[] tiles;
    public Tile[] Tiles {
        get => tiles;
    }

    public Triplet(Tile[] tiles) {
        this.tiles = new Tile[3];
        Array.Copy(tiles, this.tiles, 3);
    }

    public static implicit operator Triplet(Tile[] tiles) {
        return new Triplet(tiles);
    }

    public Tile this[int i] {
        get => tiles[i];
        set => tiles[i] = value;
    }

    public override string ToString() => $"[{string.Join(' ',
tiles)}]";
}
```

rubiks-cube/Tile.cs (lines 3-11)

```
public struct Tile {
    public Color Color { get; set; }

    public Tile(Color color) {
        Color = color;
    }

    public override string ToString() => $"{Color}";
}
```

rubiks-cube/Color.cs (lines 3-11)

```
public enum Color : byte {
    I = 0, // Invalid
    W,
    B,
    R,
    G,
    O,
    Y,
}
```

We will leave it to the readers to go through this sample code and understand the overall APIs and the implementations.

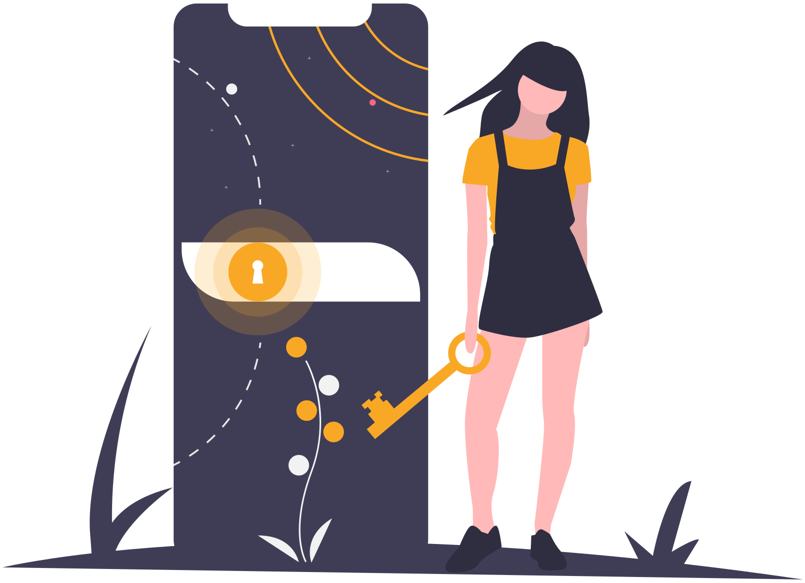
Clearly, this is not the only way, and the readers are encouraged to look for other possible way to represent the Rubik’s Cube as an "object" (in the sense of the OOP).

Chapter 29. Key-Value Store

29.1. Problem

There are many storage or database systems that store data in a dictionary format, that is, as a key-value mapping.

Let’s implement a simple key-value store.



29.2. Discussion

In the .NET world, libraries are shared through "NuGet packages".

There is an interesting library for parsing command line arguments (flags and what not). www.nuget.org/packages/System.CommandLine. We will use this library for the project of this lesson.

In order to be able to use a third-party library, you need to "install" it in your project.

If you use the `dotnet` CLI, then you can use the `dotnet add` command. We used `dotnet add` before to add a reference to an internal library project in an earlier lesson.

In this project we use `dotnet add package` command.

```
dotnet add package --help
Usage: dotnet add <PROJECT> package [options] <PACKAGE_NAME>

Arguments:
  <PROJECT>        The project file to operate on. If a file is not
specified, the command will search the current directory for one.
  <PACKAGE_NAME>   The package reference to add.

Options:
  -h, --help                          Show command line help.
  -v, --version <VERSION>             The version of the package to
add.
  -f, --framework <FRAMEWORK>         Add the reference only when
targeting a specific framework.
  -n, --no-restore                    Add the reference without
performing restore preview and compatibility check.
  -s, --source <SOURCE>               The NuGet package source to use
during the restore.
  --package-directory <PACKAGE_DIR>   The directory to restore
packages to.
  --interactive                       Allows the command to stop and
wait for user input or action (for example to complete
authentication).
  --prerelease                        Allows prerelease packages to
be installed.
```

For example,

```
dotnet add package System.CommandLine --version 2.0.0-beta1.21216.1
```

Regardless of how you add the package reference, the project XML file ends up getting the following line:

```
<ItemGroup>
  <PackageReference Include="System.CommandLine" Version="2.0.0-beta1.21216.1" />
</ItemGroup>
```

> The version number of the library might be different when you read this book. Check the most recent version from the official NuGet website.

For our key-value store project, we will borrow some CLI commands from Redis, redis.io/commands, for inspiration. ☺

In particular, we will try to implement the following methods:

- ping
- keys
- get key
- set key value
- del key
- exists key

Note that Redis is baed on the client-server architecture. That is, a client program and a server program run in separate processes, possibly on different machines, and they communicate over the network, e.g., over the local network or over the Internet.

For the project in this lesson, however, we will create a single program that includes both "client" and "server" components.

In theory, we can refactor the code to create two separate programs, one for the client and the other for the server. In order to do that, we will need to add a network communication layer between the client and the server. For the sample code, we merely use the internal API calls.

29.3. Sample Code Snippets - *key-value-store*

Note that we set the command name to `kvstore` in the project XML file for this program:

```
<PropertyGroup>
  <AssemblyName>kvstore</AssemblyName>
</PropertyGroup>
```

Here’s the main (client) program:

key-value-store/Program.cs (lines 4-8)

```
IServer<string> server = new Server();
var client = new Client(server);

var rootCommand = client.BuildRootCommand();
await rootCommand.InvokeAsync(args);
```

It reads the command line argument, and passes it to the "client".

Note that this program has a rather similar structure as the sample code of the previous lesson. We use an abstraction for the "server" component using an interface, `IServer<string>`.

A "client" instance is constructed by taking a concrete implementation of this interface type `IServer<string>` as an argument.

> Although it is beyond the scope of this book, this kind of pattern can be used for "dependency injection" (or, DI for short), if needed. The readers are encouraged to look up on this topic, if interested.
>
> As stated, software development requires a lot of different skills and knowledge beyond mere "programming".

The `Client` encapsulate the functionalities of the "CLI client".

The `System.CommandLine` library more or less requires the program to start as follows:

```
await rootCommand.InvokeAsync(args);
```

We are using the async version of the `Invoke()` method here. As stated, the generated `Main()` method of this program will be an async version. The `rootCommand` is a variable for type `System.CommandLine.RootCommand`.

The client supports a number of different "sub-commands" below the root command.

key-value-store/Client.cs (lines 5-33)

```
public sealed class Client {
    private IServer<string> server;

    public Client(IServer<string> server) => this.server = server;

    public RootCommand BuildRootCommand() {

        var rootCommand = new RootCommand(description:
            "A demo app to store key-value data.")
        {
                new Option<bool>(new string[]{"--reset-all"}, "Delete all data and start fresh"),

                new PingCommand(server),
                new GetCommand(server),
                new SetCommand(server),
                new KeysCommand(server),
                new ExistsCommand(server),
                new DelCommand(server),
            };
        rootCommand.Handler = CommandHandler.Create<bool>(RootCommandHandler);

        return rootCommand;
    }

    private void RootCommandHandler(bool resetAll) {
        // Placeholder for now.
        Console.WriteLine($"resetAll: {resetAll}");
    }
}
```

A series of sample commands:

key-value-store/Commands/Ping.cs (lines 5-20)

```
public sealed class PingCommand : Command {
    private IServer<string> server;

    public PingCommand(IServer<string> server,
        string name = "ping",
        string description = "Ping the server") : base(name, description) {
        this.server = server;

        Handler = CommandHandler.Create(Ping);
    }

    public void Ping() {
        var result = server.Ping();
        Console.WriteLine($"Ping: {result}");
    }
}
```

key-value-store/Commands/Get.cs (lines 5-26)

```
public sealed class GetCommand : Command {
    private IServer<string> server;

    public GetCommand(IServer<string> server,
        string name = "get",
        string description = "Get the value of a key")
        : base(name, description) {
        this.server = server;

        AddOption(new Option<string>(new[] { "--key", "-k" },
"Key"));
        Handler = CommandHandler.Create<string>(Get);
    }

    public void Get(string key) {
        var value = server.Get(key);
        if (value is not null) {
            Console.WriteLine($"Value {value} for {key}");
        } else {
            Console.WriteLine($"Item for {key} does not exist.");
        }
    }
}
```

key-value-store/Commands/Set.cs (lines 5-26)

```
public sealed class SetCommand : Command {
    private IServer<string> server;

    public SetCommand(IServer<string> server,
        string name = "set",
        string description = "Set the value of a key to the given
value") : base(name, description) {
        this.server = server;

        AddOption(new Option<string>(new[] { "--key", "-k" },
"Key"));
        AddOption(new Option<string>(new[] { "--value", "-v" },
"Value as a string"));
        Handler = CommandHandler.Create<string, string>(Set);
    }

    public void Set(string key, string value) {
        var oldValue = server.Set(key, value);
        if (oldValue is not null) {
            Console.WriteLine($"Replaced {oldValue} with {value} for {key}.");
        } else {
            Console.WriteLine($"Created a new item for {key} with value {value}.");
        }
    }
}
```

key-value-store/Commands/Keys.cs (lines 5-20)

```
public sealed class KeysCommand : Command {
    private IServer<string> server;

    public KeysCommand(IServer<string> server,
        string name = "keys",
        string description = "List all keys") : base(name, description) {
        this.server = server;

        Handler = CommandHandler.Create(Keys);
    }

    public void Keys() {
        var keys = server.Keys();
        Console.WriteLine($"All keys: {string.Join(", ", keys)}");
    }
}
```

key-value-store/Commands/Exists.cs (lines 5-24)

```
public sealed class ExistsCommand : Command {
    private IServer<string> server;

    public ExistsCommand(IServer<string> server,
        string name = "exists",
        string description = "Returns true if a key exists, false otherwise") : base(name, description) {
        this.server = server;

        AddOption(new Option<string>(new[] { "--key", "-k" }, "Key"));
        Handler = CommandHandler.Create<string>(Exists);
    }

    public void Exists(string key) {
        if (server.Exists(key)) {
            Console.WriteLine($"{key} exists");
        } else {
            Console.WriteLine($"{key} does not exist");
        }
    }
}
```

key-value-store/Commands/Del.cs (lines 5-25)

```
public sealed class DelCommand : Command {
    private IServer<string> server;

    public DelCommand(IServer<string> server,
        string name = "del",
        string description = "Delete the entry for a given key") : base(name, description) {
        this.server = server;

        AddOption(new Option<string>(new[] { "--key", "-k" },
"Key"));
        Handler = CommandHandler.Create<string>(Del);
    }

    public void Del(string key) {
        var oldValue = server.Del(key);
        if (oldValue is not null) {
            Console.WriteLine($"Deleted {oldValue} for {key}.");
        } else {
            Console.WriteLine($"Item for {key} does not exist.");
        }
    }
}
```

Note that the "client" commands rely on the server implementations. That is, the `Server` class provides the actual implementations for these commands:

key-value-store/Server.cs (lines 3-19)

```
public sealed class Server : IServer<string> {
    private Store<string> store;

    public Server() => store = new();

    public string? Ping() => "Pong";

    public string? Get(string key) => store.Get(key);

    public string? Set(string key, string value) => store.Set(key,
value);

    public string[] Keys() => store.Keys();

    public bool Exists(string key) => store.Exists(key);

    public string? Del(string key) => store.Del(key);
}
```

In this demo, we just use `Dictionary<string,T>` in memory as a "database":

key-value-store/Store.cs (lines 3-31)

```
public sealed class Store<T> {
    private readonly Dictionary<string, T> map;

    public Store() => map = new();

    public T? Get(string key) => map.ContainsKey(key) ? map[key] : default;

    public T? Set(string key, T value) {
        T? oldValue = default;
        if (map.ContainsKey(key)) {
            oldValue = map[key];
        }
        map[key] = value;
        return oldValue;
    }

    public string[] Keys() => map.Keys.ToArray();

    public bool Exists(string key) => map.ContainsKey(key);

    public T? Del(string key) {
        T? oldValue = default;
        if (map.ContainsKey(key)) {
            oldValue = map[key];
            map.Remove(key);
        }
        return oldValue;
    }
}
```

As indicated, different server implementations can be used through the abstraction based on the `IServer` interface.

key-value-store/Common/IServer.cs (lines 3-10)

```
public interface IServer<T> {
    string? Ping();
    T? Get(string key);
    T? Set(string key, T value);
    string[] Keys();
    bool Exists(string key);
    T? Del(string key);
}
```

Author’s Note

Request for Feedback

The author is always looking to improve the book.

If you have any suggestions, then please let us know. We, and the future readers, will really appreciate it.

It can be anything from simple typos, unclear sentences, and formatting errors to any bugs in the sample code and maybe downright incorrect explanations. Here’s the author’s email:

- harry@codingbookspress.com.

The author will try to correct the errors as soon as possible, as needed.

Thank you!

Part IV: Final Projects

Life’s like a movie, write your own ending. Keep believing, keep pretending. -Jim Henson

Chapter 30. Reverse Polish Calculator

30.1. Project

As one of the three final projects, we will work on implementing a calculator in this lesson, which can process an arbitrary arithmetic expression. For example, let's consider the following input:

```
(1 + 3) * (6.0 / 2) - 3.5
```

It includes the four different arithmetic operations, addition (+), multiplication (*), division (/), and subtraction (-). The parentheses are used to change the precedence in mathematical expressions. This expression evaluates to 8.5, as you can easily verify.

Writing a program to evaluate a general arithmetic expression like this is a relatively complicated problem. We will need to write a "parser" to parse an input expression.

> If any of the (ambitious) readers is interested, then this can be done by writing a simple "recursive descent parser". We will leave it to the readers. If you are learning programming without computer science background, then this can be a very interesting, and challenging, problem. You will end up learning *a lot* by writing a simple calculator program.

For this project, however, we will tackle a slightly simpler problem. The computation can be simplified by using what we call the "postfix notation", aka

Reverse Polish notation. Here’s the link to a Wikipedia article: Reverse Polish notation [https://en.wikipedia.org/wiki/Reverse_Polish_notation].

The above input expression in the standard "infix notation" can be written as follows in the postfix notation:

```
1 3 + 6.0 2 / * 3.5 -
```

Note that the postfix notation does not require any use of parentheses. Note also that this simple notation can only support binary operations (that is, those with two operands). Take your time and see if you can figure out how to read this expression. We will go over this together in the next section.

Given an input like this, our program will need to produce an answer like this:

```
The result is 8.5
```

30.2. Design Suggestion

The easiest way to implement a Reverse Polish calculator is using "stacks". A stack is a LIFO data structure (last in first out), and it generally supports interfaces for adding (to the end/top of the stack) and for removing items (from the end/top): Stack (abstract data type) [https://en.wikipedia.org/wiki/Stack_(abstract_data_type)].

Let's look at how the postfix notation works, using the above example. We will just use a stack to illustrate this concept instead of going through this twice, once without a stack and the second time with a stack.

1. When we see a number, `1` in this case, we push it into a stack. At this point, we have one item in the stack.
2. When we see another number, `3`, we push it again into the stack. Now, we have `3` and `1` in the stack, from top to bottom.
3. When we see a binary operator, `+` in this case, we pop two elements at the top of

the stack, that is, `3` and `1` in this example, and compute the expression, `3 + 1`. The result `4`, a number, is then pushed back into the stack. At this point, the content of the stack is this one item `4`.

4. Now, we have `6.0` in the input, another number. This is pushed into the stack. The stack content: `{6.0, 4}`.
5. Then, the number `2` is also pushed into the stack. The stack content: `{2, 6.0, 4}`.
6. When we see the binary operator `/`, we pop the two elements at the top, and compute the expression, `6.0 / 2`. Note the order. The value of this expression is `3.0`. This number is pushed back into the stack. The stack content at this point: `{3.0, 4}`.
7. The next token is another operator `*`. Hence, we pop the top two elements and do the operation, `3.0 * 4`, whose result is put back into the stack. Now, the stack contains only one item, `12.0`.
8. The next number `3.5` from the input is pushed into the stack. At this point, the stack content is `{3.5, 12.0}`.
9. We then compute `12.0 - 3.5` since the next token is a binary operator `-`, and we put the result `8.5` back into the stack.
10. At this point, we have processed all the input tokens, and the item at the top of the stack, `8.5`, is the result of the whole expression.

Implementing a Reverse Polish calculator is straightforward.

- First, we read the input expression.
- Next, we "tokenize" this expression. That is, we divide it into a sequence of numbers and binary operators. For simplicity, we will just assume that all tokens are separated by at least one whitespace, e.g., spaces and/or tabs.
- Then, every time we see a number, push it into a stack.
- And, every time we see an operator, we pop the top two items from the stack

and compute the binary expression. The result is pushed back into the stack.

- When we have no more tokens to process, the top element in the stack is the result of the input expression.

You can use a stack data structure from the .NET library, or you can create your own stack type using what we have learned in this book.

Here's an example:

```
public sealed class Stack<T> where T : notnull {
    private readonly List<T> list;

    public Stack() => list = new();

    public int Size => list.Count;

    public T Push(T tok) {
        list.Add(tok);
        return tok;
    }

    public T? Pop() {
        if (list.Count > 0) {
            var tok = list[^1];
            list.RemoveAt(list.Count - 1);
            return tok;
        }
        return default;
    }

    public bool TryPop(out T? tok) {
        if (list.Count > 0) {
            tok = list[^1];
            list.RemoveAt(list.Count - 1);
            return true;
        }
        tok = default;
        return false;
    }

    public T PopOrDie() {
        if (list.Count > 0) {
            var tok = list[^1];
            list.RemoveAt(list.Count - 1);
            return tok;
        }
        throw new IndexOutOfRangeException();
    }

    public (bool, T?) PopSoft() {
        if (list.Count > 0) {
            var tok = list[^1];
            list.RemoveAt(list.Count - 1);
            return (true, tok);
        }
        return (false, default);
    }
}
```

This stack implementation uses `System.Collections.Generic.List<T>` to store the internal data (through "composition"), and it exposes the conventional `Push()` and `Pop()` methods as public API.

This example shows four different possible implementations of the `Pop()` method. (Obviously, you will only need one.) The implementation of the `Push()` method is straightforward. But, for `Pop()`, we need to consider an edge case when the stack does not contain any data.

Traditionally, we could have indicated this special situation by returning the `null` value, if `T` is a reference type, as shown in the first `Pop()` method. This implementation is not as convenient when `T` is a value type since the type's `default` value might be a valid value for the stack data.

Another way is just to throw an exception. This example implementation is shown in the `PopOrDie()` method. The caller will need to `try` and `catch` any possible exceptions. As alluded throughout the earlier parts of this book, the modern trend in C# programming is not to use exceptions, when avoidable. (And, avoid the `null` value, if possible.)

A "better" alternative might be using the `TryXXX` pattern. This is shown in the implementation of the `bool TryPop(out T? tok)` method. We have seen a number of such "Try" methods earlier in the book. We only use the `tok` out parameter value if the method returns `true`.

The `PopSoft()` method essentially follows the same pattern, but it uses a tuple return value instead of using the `out` parameter.

Again, the choice is yours. The C# programming style is changing, and there is no "right" or "wrong" way. It is really a matter of preference at this point.

> ℹ If you implement a calculator, then you will likely define a "token" type (for numbers and operators) and it is this token type that will be stored in the stack (the `T` type parameter). In the example above, the nullable `T?` type is an artifact of the generic implementation. You will likely use a non-nullable record/struct as a token type, but there is no easy way to specify that as a generic type constraint, as of the current version of C#.

30.3. Sample Implementations

We will leave the implementation to the readers. After all, this is a "hands-on" project. ☺ An example program is included in the appendix: [appendix-code-listing-part4].

> No cheating! 😃😃😃
>
> There are two implementations. `reverse-polish-1` and `reverse-polish-2`. They are more or less equivalent, but one sample program uses a property `TokType` to distinguish different token types. The other one uses the object types, and the polymorphism, to achieve the same.
>
> One sample code is not necessarily "better" than the other. But, there will be situations where one style is preferred over the other. It will be instructive to compare these two programs.

> Not all input sequences are valid expressions in the postfix notation. The sample program, either version, does not do much error handling when it encounters an invalid input expression. It will be a good exercise to think of a good error handling method for this problem.

Chapter 31. Web To-Do List

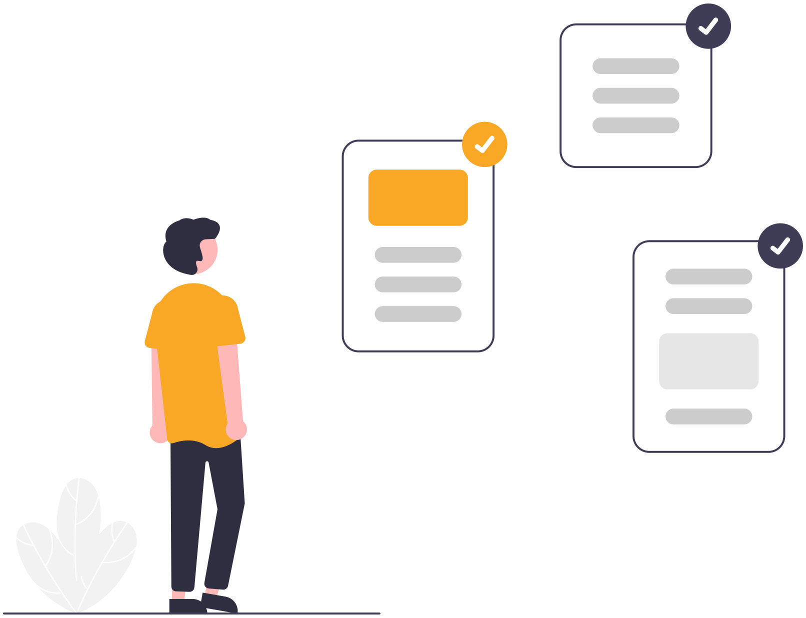
31.1. Project Idea

Creating a "to do list" app is one of the most popular projects for beginning programmers.

Here’s a list of requirements for this final project:

1. Create a Web server app that stores to-do list items in memory.
2. Create a CLI client which connects to the server via HTTP.
3. The client app supports the following commands:
    - Create a to-do item.
    - List all current to-do items.
    - Delete an item.
    - Update an item.

> ℹ There is no more sample code for the remaining final projects. This is the time for you to test your C# skills, and to have fun! 🚀



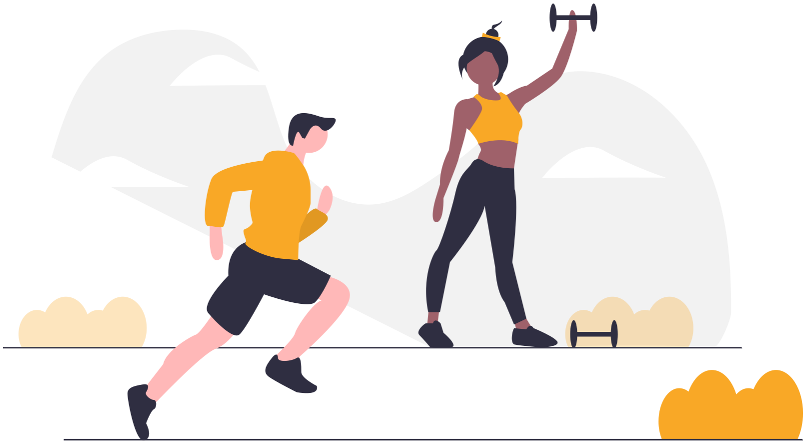
Chapter 32. Rubik’s Cube Challenge

32.1. Final Final Projects

Here’s one final "very ambitious" project. Or, more like three projects.

Let’s "solve" Rubik’s cube using computers!!!

We will try this in three different ways.



32.2. Random Tries

If we try various moves enough times on a cube, then will it reach the target state? Try a series of random rotations, in an infinite loop, and check if the cube is in the perfectly ordered state.

32.3. Optimization

We can use a simple optimization technique to see if it works in solving the Rubik’s cube problem. Based on the arrangement of the tiles, assign a score for each configuration. That is, the more the cube is ordered, the higher score it will have. Try an optimization technique using this score value.

32.4. Rubik's Cube Algorithm

In fact, a (semi-)deterministic algorithm exists to solve the Rubik's cube, Rubik's Cube 3x3 guide [https://www.rubiks.com/en-uk/rubiks-cube-3x3-guide]. Try implementing this algorithm in C#.

> These are not easy projects for beginners, or for anyone. Give them a try. Just remember. The whole point is *having fun*. Good luck! 👍

Author's Note

Final Remarks

Congratulations! You made it!

It is not easy to read a technical book like this from beginning to end, regardless of your skill levels. This is a big achievement. Congrats!

As stated, knowledge is familiarity. The more you read, and the more you practice, the better you will become. In any art. Especially, in the art of programming.

Hope you had as much fun reading this book as I did writing it.

Index

Credits

Images

All drawings used in this book are taken from undraw.co, an amazing service with an amazing open source license. Many thanks to the creator of the site: twitter.com/ninaLimpi!

Icons

All emoji icons used in this book are from fontawesome.com. Fontawesome is a very popular tool, probably used by almost everyone who does Web or mobile programming.

Typesetting

Here's another absolutely fantastic software, asciidoctor.org, which is used to create an ebook as well as paperback versions of this book. AsciiDoc [https://asciidoc.org] is like Markdown on steroid. You can follow them on Twitter: twitter.com/asciidoctor.

Other Resources

The author has relied on many resources on the Web in writing this book, in particular, .NET documentation [https://docs.microsoft.com/en-us/dotnet/]. If the book includes any material from these resources, then the copyright of those content belong to the respective owners.

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/]

Mini Programming Language References

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

> Already published, or to be published, throughout 2023

- Go Mini Reference
- Modern C# Mini Reference
- Python Mini Reference
- Typescript Mini Reference
- Rust Mini Reference
- C++20 Mini Reference
- Modern Java Mini Reference
- Julia Mini Reference
- Javascript Mini Reference
- Haskell Mini Reference
- Scala 3 Mini Reference

- Lua Mini Reference