Windows Applications Programming

Using Windows Forms and Microsoft .NET

LIVIU-ADRIAN COTFAS

Windows Applications Programming

Using Windows Forms and Microsoft .NET

EDITURA UNIVERSITARĂ
Bucureşti

Colecţia ŞTIINŢE EXACTE

Redactor: Gheorghe Iovan
Tehnoredactor: Ameluţa Vişan
Coperta: Monica Balaban

Editură recunoscută de Consiliul Naţional al Cercetării Ştiinţifice (C.N.C.S.) şi inclusă de Consiliul Naţional de Atestare a Titlurilor, Diplomelor şi Certificatelor Universitare (C.N.A.T.D.C.U.) în categoria editurilor de prestigiu recunoscut.

Descrierea CIP a Bibliotecii Naţionale a României
COTFAS, LIVIU-ADRIAN
Windows applications programming / Liviu-Adrian Cotfas. –
Bucureşti : Editura Universitară, 2021
Conţine bibliografie
ISBN 978-606-28-1295-9

004

DOI: (Digital Object Identifier): 10.5682/9786062812959

Contents

PREFACE

The book serves as lecture support for the Windows Applications Programming course taught at the Bucharest University of Economic Studies. It can also be a relevant reference for software developers focusing on developing Windows applications and for any other person interested to gain further insights into building Windows Forms applications.

The code associated to the examples in this book is available on GitHub in the repository https://github.com/liviucotfas/ase-windows-applications-programming .

1. C# & .NET FRAMEWORK BASICS

1.1. Objectives

- understand the basic structure of a Console application.
- decompile a .NET application.
- read and write information from/to the Console.

1.2. Introduction

C# has been developed around 2002 by Microsoft, as a general-purpose, type-safe, platform neutral, object-oriented programming language [1]. The core objective of the language is developer productivity. The language follows the C and C++ syntax and has been designed to allow the development of a wide variety of applications using the Microsoft .NET Framework. The .NET Framework is composed from a Comon Language Runtime (CLR) and a comprehensive set of libraries.

The CLR is the runtime that handles the execution of managed code in the context of the .NET Framework and that provides a layer of abstraction between the application and the operating system. Managed code is represented in Intermediate Language (IL), into which the applications written in one of the managed languages are converted when compiled. Alongside C#, other managed languages that get compiled into managed code include F#, Visual Basic .NET, Managed C++, Delphi .NET and J#. The IL code is stored, together with metadata, in assemblies, which can be either executable files (.exe) or libraries (.dll). The code in the assemblies is

CPU independent. Thus, when the CLR loads an assembly, it prepares it for execution by converting the IL code into native code, such as x86. The conversion is performed by the Just-In-Time (JIT) compiler component of the runtime. In order to improve performance, the JIT compiler does not convert the entire assembly to native code, but the conversion rather happens on an as-needed basis. Afterwards, when a method is called, the runtime first checks if the code for the method has already been converted and placed in the cache.

The CLR also manages the code execution, the user-level security, the automatic allocation and release of memory, while also providing structured exception handling.

In order to facilitate the development of .NET applications using different managed languages, the CLR employees a Common Type System (CTS) and a Common Language Specification (CLS). CTS is used to describe all the possible data types and programming constructs supported by the runtime. CLS is a subset of CTS defining a subset of features that should be supported by any managed language designed for .NET. Thanks to the fact that code written in any managed language is converted to IL, it becomes possible to mix in the same application classes written using different managed languages.

While in the past .NET applications have commonly been restricted to the Windows operating system, with the introduction of .NET Core, it has become possible to run certain types of .NET applications on Linux, macOS, Android, iOS, tvOS and watchOS as well. With the help of Xamarin, .NET applications can also run on Samsung Smart TVs, running the Tizen operating system [2]. The supported processor architectures are arm32, arm 64, x86 and x64. The framework is capable of providing access to platform-specific capabilities, such as the APIs provided by different operating systems.

The main types of applications that can be developed using C# and the .NET framework are shown in Figure 1 and include web applications, mobile applications, desktop applications, cloud applications, games and machine learning applications.

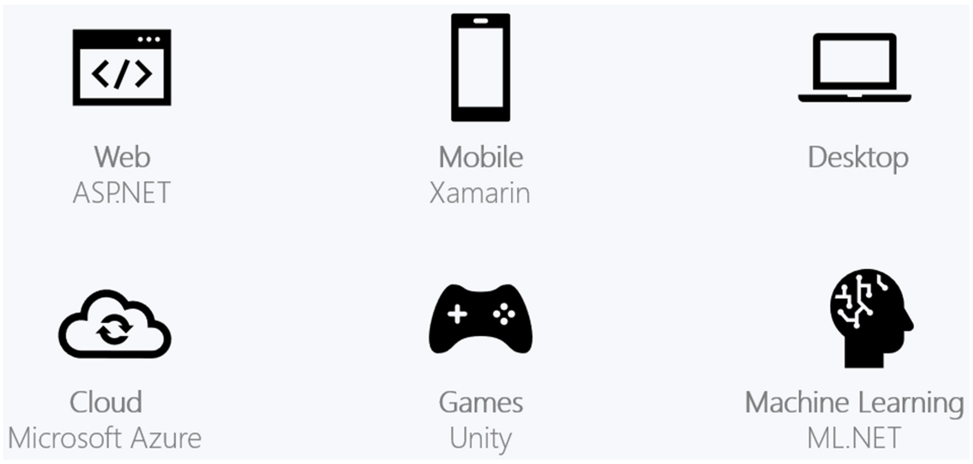

Figure 1. Types of applications that can be developed using C# and .NET

The full API can be browsed online, by accessing the website docs.microsoft.com/en-us/dotnet/api . Additionally, a wide variety of official samples are available at https://code.msdn.microsoft.com .

While .NET applications can be written using any text editor, the recommended integrated development environments are Visual Studio (Windows), Visual Studio Code (Windows, macOS, and Linux) and Visual Studio for Mac (macOS).

1.3. Comparison with C++

In C#, global methods and variables are not supported. Thus, methods and variables must be contained within a class or a structure.

The Main method, the point where the execution of the program begins, is capitalized in C# [3] and needs to always have the static modifier, as shown in Figure 2.

```
private static void Main(string[] args)
{
    //HelloWorld application
    Console.WriteLine("Hello World!");
    Console.ReadLine();
} //end main
```

Figure 2. Main method in a C# application

Multiple inheritance has not been implemented in NET. This choice has been made in order to avoid various issues related to this development approach [3]. Thus, a class can inherit from only one base class, but can implement any number of interfaces, as shown in Figure 3 [4].

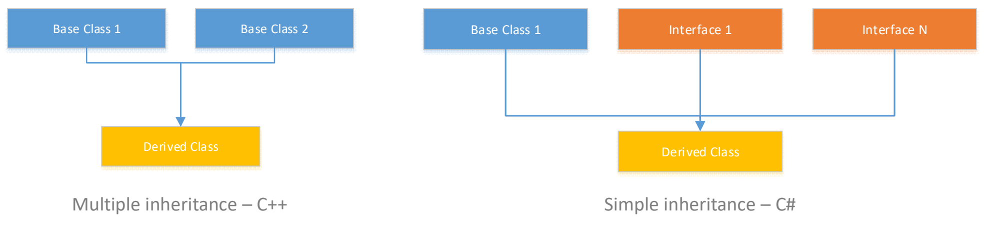

Figure 3. Multiple and simple inheritance

While in the case of C/C++ the memory allocation in the heap must be handled by the software developer, .NET implement a mechanism called Garbage Collector (GC) which automatically allocated and releases memory.

1.4. First C# Program

The execution of every C# application begins in a method called Main. There should only be one such method insider an application. Observations:

- in C# the Main method is capitalized, while in the case of Java, lowercase main is used;
- note the static modifier which has a similar behavior to C++.

Activity:

1. Create a new Microsoft .NET Project in Visual Studio;
2. Update the content of the Program class in order to match the code included in Figure 4

```
using System; //referenced namespace
namespace NameSpaceProgram
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            //HelloWorld application
            Console.WriteLine("Hello World!");
```

```
                Console.ReadLine();
            } //end main
        }//end class
} //end namespace
```

Figure 4. Hello World application written in C#

The fact that the assemblies contain almost all the original source code facilitates their inspection.

Activity:

1. Download dotPeak from https://www.jetbrains.com/decompiler/.
2. Decompile the HelloWorld application.

1.5. Reading and Writing using System.Console

Data can be displayed in the operating system console with the help of the Sysstem.Console[1] class. The Write and WriteLine methods of this class can be used in order to write data to the standard output stream. Reading data from the standard input stream can be performed with the help of the Read and ReadLine methods.

Several approach for displaying the values of two variables are shown in Figure 5.

```
int foo = 10;
int bar = 20;
//String concatenation:
Console.WriteLine("foo: " + foo + " bar: "+ bar);
//Composite formatting:
Console.WriteLine("foo: {0} bar: {1}", foo, bar);
Console.WriteLine("{1}, {0}, {2}", 10, 20, 30);
//String interpolation: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated
Console.WriteLine($"{foo}{bar}");
//Format
Console.WriteLine("c format: {0:c}", foo);
```

Figure 5. Writing data to the strandard output stream

[1] https://docs.microsoft.com/en-us/dotnet/api/system.console

1.6. Specifying an Application Error Code

The Main() method in a console application can return either int or void. An int return type can be used in order to return a value to the application that has called the console application [3].

Activity:

1. Replace the code from the previous activity with the following one.

```
static int Main()
{
    Console.WriteLine("Hello World!");
    Console.ReadLine();
    // you can return an error code
    return -1;
}
```

2. Create the following batch script and call it from the Console.

```
@echo off

NameofTheExecutableFile

@if "%ERRORLEVEL%" == "0" goto ok
:error
echo There was an error!
echo return value = %ERRORLEVEL%
goto end
:ok
echo Everything ok!
echo return value = %ERRORLEVEL%
goto end
:end
echo All Done.
```

1.7. Processing Command-Line Arguments

The Main() method can receive command line arguments as a string array. The argument is optional and can be omitted if the command line arguments are not needed [3].

Activity:

1. Replace the code from the previous activity with the following one.

```
public static void Main(string[] arguments)
{
    for(int i=0; i< arguments.Length; i++)
    {
        Console.WriteLine(arguments[i]);
    }
}
```

2. Run the application using the Console as follows.

```
NameofTheExecutableFile.exe /argument1 -argument2
```

2. DATA TYPES

2.1. Objectives

- understanding the data types hierarchy.
- understanding the role of the System.Object class.
- understanding the particularities of the System.String class.
- working with arrays.
- working with multidimensional arrays.

2.2. Data Types

All the types[2], including built-in simple types, such as bool or int are ultimately derived from a common base type, which is System.Object, as shown in Figure 6. The unified hierarchy of types provided by the framework constitutes the Common Type System (CTS).

The types in CTS can be classified as either value types or reference types [3]. Value types include enumerations and structures and are derived from System.ValueType.

Value types[3] include the built-in numeric types, as well as other built-in types, such as System.Boolean. Their value is stored on the stack and the variables are removed when the execution of the application exits the scope in which the variable has been declared. Since they are allocated on the

[2] https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/
[3] https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/#value-types

stack, and not in the heap, such variables can be created and destroyed very quickly.

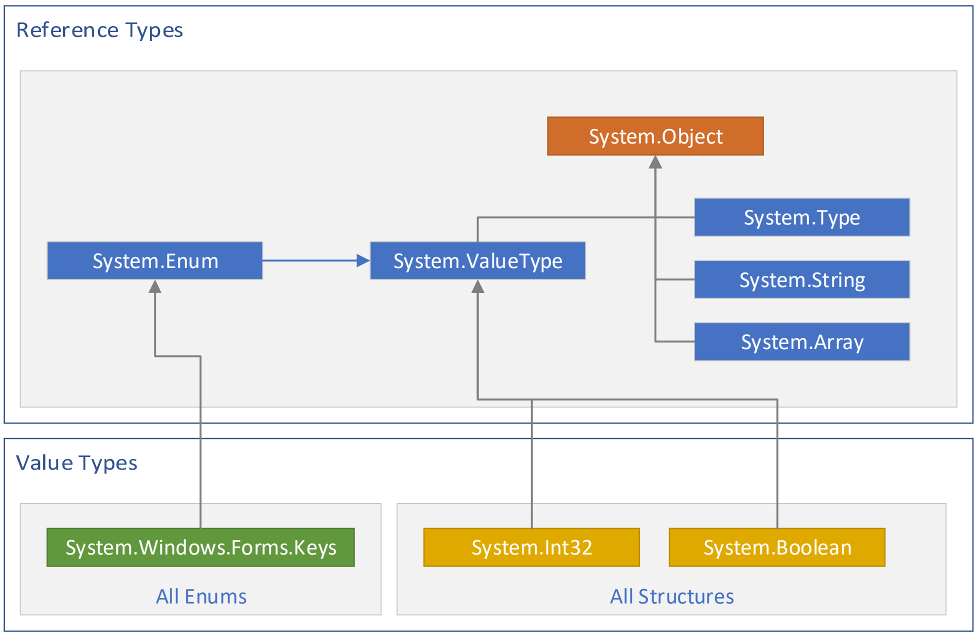

Figure 6. Reference and Value Types

Reference types[4] are declared using the keywords class, interface and delegate and can inherit from any type, with the exception of the ones derived from System.ValueType. Compared to value types, the moment when the memory is deallocated is influenced by many factors [3]. A comparison between characteristics of value types and reference types is provided in Table 1.

Table 1. Comparison between Value Types and Reference Types

	Value Types	**Reference Types**
Allocation	Allocated on the stack.	Allocated in the managed heap.
Variable lifetime	Can be created and destroyed very quickly. The lifetime is determined by the defining scope.	Have a lifetime that is determined by a large number of factors. They are destroyed when they are garbage collected.

[4] https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/#reference-types