Pro .NET Framework with the Base Class Library

Understanding the Virtual Execution System and the Common Type System

Roger Villela

Apress®

Pro .NET Framework with the Base Class Library

Understanding the Virtual Execution System and the Common Type System

Roger Villela

Apress®

Pro .NET Framework with the Base Class Library: Understanding the Virtual Execution System and the Common Type System

Roger Villela
São Paulo, Brazil

ISBN-13 (pbk): 978-1-4842-4190-5 ISBN-13 (electronic): 978-1-4842-4191-2
https://doi.org/10.1007/978-1-4842-4191-2

Library of Congress Control Number: 2019932018

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: Matthew Moodie
Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

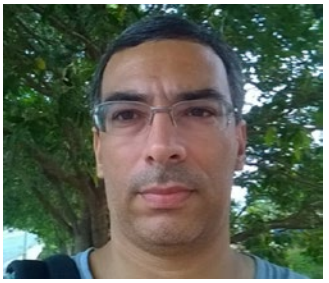
Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4190-5. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to my mother, Marina Roel de Oliveira.†
†From January 14, 1952, to March 17, 2017

Table of Contents

About the Author



Roger Villela is a software engineer with 30 years of experience in the industry. He now works as a professional technical educator in his own firm that specializes in the inner workings of the following Microsoft development platforms: the Windows operating system, the Universal Windows Platform (UWP), and the .NET Framework. He has worked with various tech companies and now bases his work on Microsoft Visual Studio 2017 with Microsoft Visual C++/CLI projection, Component Extensions (C++/CX) projection, C++/WinRT, Intel Parallel Studio XE with Intel C++ (Microsoft Windows), and Assembly (Intel x86/Intel x64).

About the Technical Reviewer

James Millar is a freelance .NET developer based in Bristol in the United Kingdom. He has more than ten years of experience in the IT industry. He has worked on Microsoft Azure cloud computing (PaaS and IaaS), Azure Stack, AWS Stack, DevOps, release management, and mobile application projects. He has worked with companies such as Microsoft and Sun Microsystems and has served clients based in the United States, the United Kingdom, and India.

Acknowledgments

First, I would like to thanks the team at Apress who worked with me on this book: Smriti Srivastava (acquisitions editor), Shrikant Vishwakarma (coordinating editor), Matthew Moodie (development editor), Welmoed Spahr (managing director), and James Millar (technical reviewer), Sherly Nandha (Senior Executive Project Manager), Krishnan Sathyamurthy (Production Editor), Joseph Quatela (Production Coordinator), Garrish Selvarasi (Account Manager). Is an honor and a pleasure work with such a highly professional team.

Also, thanks to my parents and a special thanks to my Mom (Marina), to my Dad (Gilberto), my two brothers (Eder and Marlos), my sister-in-law (Janaína and their parents in Fortaleza - CE), my nephew (Gabriel), my nieces (Lívia and Rafaela), my aunts, my cousins, and my uncles. Finally, I would like to thank my professional colleagues and friends who have worked with me for decades.

Introduction

Working in software engineering is a challenge and a pleasure.

This book introduces the reader to the Virtual Execution System (VES), which does the necessary work of keeping the managed environment working as smoothly as possible. That is, the VES takes care of many aspects of the inner workings of the execution environment. The Common Language Runtime (CLR) is organized into various components and technologies, and the coordination between the elements presents many challenges to the managed execution environment. The VES uses managed technologies like the Common Intermediate Language (CIL) and native technologies like the C++ programming language. This book covers aspects of both of these worlds of the execution environment.

The book uses the CIL, the C++/CLI projection, and the C# programming language to show important aspects of the VES. Fundamental built-in types are used to show how the VES works when dealing with special types such as `System.Array` and `System.String`. For example, the book explains how and why the VES recognizes arrays in vector arrays (single-dimensional) and nonvector arrays (multidimensional). In addition, the book covers a specific treatment of special intermediate instructions for working with vector arrays as well as typical instructions for dealings with nonvector arrays. Internally the VES uses C++ templates as a form of optimization for working with vector arrays of fundamental built-in types such as integers. The book also navigates through the assembly manifest metadata, metadata system, and versioning, and it explains some practices that are used by .NET class libraries that should be considered as part of our day-by-day engineering practices.

CHAPTER 1

The .NET Framework

The Microsoft .NET Framework is the official name of the group of technologies and tools designed and implemented based on the Common Language Infrastructure (CLI) specification. This chapter describes the architectural and engineering resources available in any implementation of the .NET Framework on any platform.

About the Common Language Infrastructure

The CLI is an open specification that describes executable code and a runtime environment and enables multiple high-level languages to be used on different architectural platforms without being re written.

The CLR, as the name suggests, is an implementation based on the CLI specification. This book uses the Microsoft implementation of the CLR for the sample projects and source code. So, whenever the book mentions the CLR, this refers to Microsoft .NET version 4.0 or newer running on Microsoft Windows 10 or newer. The following programming languages are used in this book:

- CIL
- C++/CLI (specialized extensions to the C++ programming language)
- C++
- C#

R. Villela, *Pro .NET Framework with the Base Class Library*, https://doi.org/10.1007/978-1-4842-4191-2_1

In the CLI specification, the functionalities are organized into four main groups.

- *Common Type System (CTS)*: The CTS is a set of data types and operations that are shared by all languages that support the CTS.
- *Metadata*: The metadata describes the program structure, enabling languages and tools to work together.
- *Common Language Specification (CLS)*: The CLS includes rules (restrictions) required for language interoperability.
- *VES*: The VES is where the code is executed and how the types are instantiated, how they interact, and how they die.

This book is focused on functionalities within the CTS and VES.

Understanding detailed information about the Metadata group is not a prerequisite for component developers and application developers; however, it is important to tool builders and compiler writers. The CLS is a subset of what is in the CTS, but the content is primarily for language designers and class library (framework) designers. So, learning about the CTS will offer you and your team a great base of knowledge when starting to work with the rules in the CLS.

For example, about the types in the CLR are available application and choose the `string` reference type, we are using one of the fundamental types available through the Base Class Library (BCL), which is the foundational library that is part of the CLI specification that is implemented by the .NET Framework and .NET Core (and is the main reason for the existence of the .NET Standard). But the `string` reference type exists only because the CTS has the fundamental built-in string type defined in it, upon which the string operations are built. In fact, the string content (the value) in any instance consists of a sequence of values of the CTS platform's built-in `char` type, which is a fundamental type in the BCL. These built-in platform types, BCL fundamental types, and any other types derived or based on them all follow the rules described by the *unified type system*. In the CLI specification, this unified type system is the CTS that describes rules about conceptual, structural, and behavioral elements that must be followed by the CLI itself and specialized tools, like compilers and runtime environments.

More details about the types in the CLR are available in Chapter 2. Table 1-1 lists the types defined by the CTS and described by the Metadata group.

Table 1-1. *Fundamental CTS Data Types*

BCL Types		CTS Types
C++/CLI	**C#**	**CIL**
System::Boolean	System.Boolean	bool
System::Char	System.Char	char
System::Object	System.Object	object
System::String	System.String	string
System::Single	System.Single	float32
System::Double	System.Double	float64
System::SByte	System.SByte	int8
System::Int16	System.Int16	int16
System::Int32	System.Int32	int32
System::Int64	System.Int64	int64
System::IntPtr	System.IntPtr	native int
System::UIntPtr	System.UIntPtr	native unsigned int
System::TypedReference	System.TypedReference	typedref
System::Byte	System.Byte	unsigned uint8
System::UInt16	System.UInt16	unsigned uint16
System::UInt32	System.UInt32	unsigned uint32
System::UInt64	System.UInt64	unsigned uint64

Fundamental CTS data types

The CLI specification includes an intermediate assembly language called CIL. Here are the necessary distinctions:

- *Intermediate language (IL)*: An IL is an abstract language used by a compiler as a step between the program code and the assembly code.
- *CIL*: This is a formal instruction set to the CIL described in the CLI specification.

When you write code using some programming language that adheres to the CLI specification, the result of the compiled code is a sequence of instructions of the CIL instruction set. These instructions are not for the hardware or processor; they are for a virtual environment. The instructions include some characteristics and functionalities of the elements in a real computer, which is exactly what the resources in the CLI specification describe. These CIL instructions are known as *intermediate*. A virtual environment includes specialized aspects based on what an advanced operating system has, such as advanced security rules, mechanisms to constantly observe our own environment, and ways to guarantee data integrity based on more flexible or disciplined rules. It also includes the ability to recognize contextuality and be dynamically extensible and expandable, interacting with different and specialized environments such as data management systems, development system, and other platforms. Finally, a virtual environment is capable of being a host and of being hosted by other environments.

We must be aware that by default Microsoft Visual Studio does not includes C++/CLI extensions as part of the installed Microsoft Visual C++ features. We should check our installation and include the support to C++/CLI to run the sample code in this book, which uses C++/CLI examples. In Listing 1-1, the sample code in C++/CLI creates an instance of the `string` reference type using an unmanaged buffer as the source. This sample code has managed code and unmanaged code.

Listing 1-1. C++/CLI Sample Code Using the Namespace System

```
void wmain( ) {

constexpr char16_t* unmanagedBuffer { u"From unmanaged to managed!!!" };

String^ managedBuffer { gcnew String( ( ( wchar_t* ) unmanagedBuffer ) ) };

Console::WriteLine( "Value: {0}", managedBuffer );

Console::ReadLine( );

  return;
};
```

Listing 1-2 shows an example of a sequence of CIL code generated from the C++/CLI sample code in Listing 1-1. All highlighted items are examples of CIL instructions.

Listing 1-2. Excerpt of CIL

```
.method assembly static int32 modopt([mscorlib]System.Runtime.
CompilerServices.CallConvCdecl)
        wmain() cil managed
{
  .vtentry 1 : 1
  // Code size       34 (0x22)
  .maxstack  2
  .locals ([0] string managedBuffer,
           [1] char* modopt([mscorlib]System.Runtime.CompilerServices.IsConst)
               modopt([mscorlib]System.Runtime.CompilerServices.IsConst)
               unmanagedBuffer)
  IL_0000:  ldnull
  IL_0001:  stloc.0
  IL_0002:  ldsflda    valuetype '<CppImplementationDetails>'.$ArrayType
                       $$$BY0BN@$$CB_S modopt([mscorlib]System.Runtime.
                       CompilerServices.IsConst) '??_C@_1DK@JJAGDPPJ@?$AAF?
                       $AAr?$AAo?$AAm?$AA?5?$AAu?$AAn?$AAm?$AAa?$AAn?$AAa?$
                       AAg?$AAe?$AAd?$AA?5@'
  IL_0007:  stloc.1
  IL_0008:  ldloc.1
  IL_0009:  newobj     instance void [mscorlib]System.String::.ctor(char*)
  IL_000e:  stloc.0
  IL_000f:  ldstr      "Value: {0}"
  IL_0014:  ldloc.0
  IL_0015:  call       void [mscorlib]System.Console::WriteLine(string, object)
  IL_001a:  call       string [mscorlib]System.Console::ReadLine()
  IL_001f:  pop
  IL_0020:  ldc.i4.0
  IL_0021:  ret
} // end of method 'Global Functions'::wmain
```

It is important to remember that this is not a one-to-one mapping between reserved words, data structures, specialized resources, or anything else in programming languages. That is, what is formalized through the instructions in the CIL is what is defined in the CLI specification and implemented by the mechanisms on the platform.

Remember, unmanaged code includes executable code and nonexecutable code that is not in the CIL and not under the management rules of the CLR environment. Erroneously, the term *unmanaged code* became synonymous with native code; this is wrong. For example, among the instructions of the CIL instruction set, there are the following instructions:

- The `cil` code implementation attribute specifies that the method declaration and implementation consist only of CIL code, that is, managed code.
- The `native` code implementation attribute specifies that the method declaration and implementation consist only of native code. This means native instructions of a specific hardware or processor platform. Currently, this functionality of the managed environment CLR implementation is used specifically as one of the base technologies to Platform Invoke (P/Invoke). P/Invoke is one of the mechanisms of the platform and is described in the CLI specification.
- The `runtime` code implementation attribute specifies that the implementation of the method is provided automatically by the runtime.

In Listing 1-3, these attributes are applied in the implementations of the methods and are mutually exclusive.

Listing 1-3. Excerpt in CIL of a Managed Method Signature

```
.method assembly static int32 modopt([mscorlib]System.Runtime.
CompilerServices.CallConvCdecl)
        wmain() cil managed
```

Two more of these attributes are available and can be combined with the others.

- `managed` is used with methods whose implementations are written using only CIL code.
- `unmanaged` describes that it is an external implementation. Currently, this code implementation attribute is used by the P/Invoke technology, but it is not restricted to be used only by that mechanism.

The following group of implementation attributes is properly categorized as code implementation attributes:

- `cil`
- `native`
- `runtime`
- `managed`
- `unmanaged`

When unmanaged code needs to be used from managed code, the `unmanaged` code implementation attribute must be applied on the method implementation. In the specific case of the P/Invoke mechanism, the use of the `unmanaged` code implementation attribute is required. Listing 1-4 shows one example of a managed code implementation that uses an unmanaged code implementation of the well-known Microsoft C Runtime (CRT)/Microsoft Universal C Runtime (UCRT) `wprintf_s` function. The `pinvokeimpl` method attribute is used to indicate that the runtime will switch from the managed state to the unmanaged state to execute the unmanaged code. Switching from the managed state to the unmanaged state, and vice versa, is performed automatically by the P/Invoke technology. In Listing 1-4, the Windows API `HeapAlloc()` function has applied the `unmanaged` and `native` code implementation attributes.

Listing 1-4. Excerpt in CIL of Unmanaged Code (Using P/Invoke to Call HeapAlloc() of Windows Memory Management)

```
.method assembly static pinvokeimpl( lasterr stdcall)
        void* modopt([mscorlib]System.Runtime.CompilerServices.CallConvStdcall)
        HeapAlloc(void* A_0,
                  uint32 modopt([mscorlib]System.Runtime.CompilerServices.
                  IsLong) A_1,
                  uint32 modopt([mscorlib]System.Runtime.CompilerServices.
                  IsLong) A_2) native unmanaged preservesig
{
  .custom instance void [mscorlib]System.Security.
  SuppressUnmanagedCodeSecurityAttribute::.ctor() = ( 01 00 00 00 )
  // Embedded native code
}
```

At this point, we understand that the CLI specification describes the CTS group, the Metadata group, the CLS and VES groups, and the CLI itself. So, what part of the managed environment is responsible for running the managed code?

About the Common Language Runtime

The Common Language Runtime, as the name suggests, is an implementation based on the CLI specification. This means it is an implementation of the Common Language Runtime that adheres to the concepts, architecture, and structural elements of what is specified in the CLI specification. Remember, in this book, we are using the Microsoft implementation of the CLR for the sample projects and code. So, whenever we mention CLR, this means Microsoft .NET 4.0 or newer running on Microsoft Windows 10 or newer.

The sample projects for the .NET Framework and their sample managed code assumes the CLR platform version 4.0 (v4.0.30319) as a minimum. We are using the .NET Framework 4.7.2 for the Microsoft Visual Studio sample projects. If we need to download a .NET Framework SDK from version 3.5 SP1 until version 4.7.2, we can find it on this official Microsoft download page:

- `www.microsoft.com/net/download/visual-studio-sdks`

All technologies and their functionalities covered are orthogonal features of the platform infrastructure. In the specific case of the CLR platform, sample projects are always based on the latest version (RTM) of the platform and tools available at the time of this writing. In this case, that includes Microsoft Visual Studio 2017 15.8.2, the .NET Framework SDK 4.7.2, and their respective runtime versions. The sample projects were created and tested on Microsoft Windows 10 April 2018 Update (aka 1803). If any sample project or explanation requires a specific version of a runtime, tool, development tool, operating system, programming language, or any related resource, this is explicitly explained.

Why are we using the .NET Framework 4.0? This version is mature and incorporates technologies that extend the capabilities and flexibility of any platform-based applications. An example is the Managed Extensibility Framework (MEF). MEF provides the ability to build models by expanding the functionality of an application already installed in the environment, without having to rewrite the application. This model is defined as an *extension*. Microsoft Visual Studio uses this technology to expand the functionalities of Microsoft Visual Studio 2017 and Microsoft Visual Studio 2015, for example.

MEF is a technology used on Microsoft Visual Studio 2010 and newer versions.

In fact, the so-called marketplace to Microsoft Visual Studio is possible thanks only to the MEF. This technology has existed since 2010 and can be used for any type of application to the .NET 4.0 platform or newer and is supported by Xamarin (Apple iOS, Android, and Apple macOS).

Now we will start exploring a couple more of the details of the CTS and VES.

About the Common Type System

When working with a sequence of bits, it is necessary to define the organization of these bits. The data represented by the bit pattern should represent the data to the type (or a contextualized type based on the data). The *type* must have a purpose and have characteristics contextually well-defined. For example, in structural terms, the data to the type must have a number of bits to define the required size and the fundamental operations that the type supports. These fundamental conceptual, structural, and behavioral characteristics create a model for what can be done and what can't be done with any of these types. In a scenario like this, with the quantity of types being expanded, it is necessary to have a set of rules so the environment works as designed and expected.

The type system model describes the necessary rules for the conceptual, structural, and behavioral characteristics.

Understanding the Fundamental Type

As an example, this section uses the Intel IA-32/x64 and Intel 64 built-in data types (or *fundamental built-in types*), some assembly instructions defined (implemented and supported) by the hardware architecture, and the contextual interpretation of the bits on the data types.

The built-in data types are those that are defined as integral elements of the platform (in this case, the Intel IA-32/x64 and Intel 64 processor hardware architecture). That means the types are integral elements of the hardware architecture and aren't defined by an external library or execution environment.

These are the fundamental types:

- Byte (8-bit) (1 byte)
- Word (16-bit) (2 bytes)
- Doubleword (32-bit) (4 bytes)
- Quadword (64-bit) (8 bytes)
- Double quadword (128-bit) (16 bytes)

Although these fundamental built-in data types are supported by a common set of assembly instructions, like MOV, they perform a common set of operations such as moving data from one place to another. Some assembly instructions support additional interpretations of the fundamental built-in data types. The purpose of these additional interpretations is to allow numeric operations to be performed, and within this context these built-in data types are viewed and manipulated as numeric data types. In the Intel IA-32/x64 and Intel 64 data types, there are integer types organized into two types: signed and unsigned. Assembly instructions like `ADD` and `SUB` can do operations on signed integers and unsigned integers, but there are assembly instructions that can do operations with only one of them.

The Organization of Fundamental Types

Here are the bits with the fundamental requirements of the hardware platform:

- Byte (8 bits)
  - Bits 7–0.
- Word (16 bits)
  - Bits 15–0.
  - The bits 15–8 form the high byte.
  - The bits 7–0 form the low byte.
- Doubleword (32 bits)
  - bits 31–0
  - The bits 31–16 form the high word.
  - The bits 15–0 form the low word.

- Quadword (64 bits)
  - bits 63–0
  - The bits 63–32 form the high doubleword.
  - The bits 31–0 form the low doubleword.
- Double quadword (128 bits)
  - Bits 127–0.
  - The bits 127–64 form the high quadword.
  - The bits 63–0 form the low quadword.

Table 1-2 presents the bits with additional interpretation and rules, based not only on the fundamental hardware requirements but on the signed and unsigned integer types.

Table 1-2. *Fundamental Data Types with Additional Interpretation*

Numeric Data Type	Description
Byte unsigned integer	All bits are used to represent the value. Values range from 0 to 255. (2^8–1)
Word unsigned integer	All bits are used to represent the value. Values range from 0 to 65,535. (2^16–1)
Doubleword unsigned integer	All bits are used to represent the value. Values range from 0 to 4,294,967,295. (2^32–1)
Quadword unsigned integer	All bits are used to represent the value. Values range from 0 to 18,446,744,073,709,551,615. (2^64–1)
Byte signed integer	The first 7 bits (6–0) are used to represent the value; the most significant bit (MSB) is used to indicate the sign. If the MSB bit has a value of 0, the number is positive. If the MSB bit has a value of 1, the number is negative. Values range from –128 to +127.

(*continued*)

Table 1-2. (*continued*)

Numeric Data Type	Description
Word signed integer	The first 15 bits (14–0) are used to represent the value; the MSB is used to indicate the sign. If the MSB bit has a value of 0, the number is positive. If the MSB bit has a value of 1, the number is negative. Values range from –32,768 to +32,767.
Doubleword signed integer	The first 31 bits (30–0) are used to represent the value; the MSB is used to indicate the sign. If the MSB bit has a value of 0, the number is positive. If the MSB bit has a value of 1, the number is negative. Values range from –2^31 to +2^31–1.
Quadword signed integer	The first 63 bits (62–0) are used to represent the value, the MSB is used to indicate the sign. If the MSB bit has a value of 0, the number is positive. If the MSB bit has a value of 1, the number is negative. Values range from –2^63 to +2^63–1.

The Work of the Fundamental Types in the CTS

The CTS supports types that describe *values* and types that specify *contracts* (behaviors that the type supports). The support for these types must be present in an implementation of the CLR. The support of these two types comes from one of the principles of the CTS, which is to support object-oriented programming and procedural and functional programming languages.

A *value* is a bit pattern used to represent types such as integer numbers and floating-pointing numbers.

For example, every C++ nonobject in the CTS is described and recognized as a *value*. In the following line of C++ code, the `unsigned int` type stands for a 32-bit (4 bytes) number.

```
constexpr uint32_t MaxItems{ 100ui32 }; /* C++ code. */
```

The following examples in C++/CLI and C# declare variables of the `UInt32` value type (and not a simple value).

```
constexpr UInt32 Limit{ 72ui32 }; /* C++/CLI. */

const uint Limit = 72; // C# code.
```

A *value* type is not an object type, but it is defined using a class definition (declaration and implementation).

Remember that this is the way of working defined by the CTS and supported by the VES in the CLR. From the perspective of the type system and execution environment, it is necessary that an object be declared, defined, and implemented to work within the CLR. Why is that?

Table 1-3 lists the fundamental built-in types defined by the CTS. You can see that there is a root object type that is accessible through the `object` keyword of the CIL. So, programming languages, such as C#, C++/CLI, F#, VB.NET and others, can access this root object type of the platform. There is a library of fundamental types that is part of the CLI specification. This foundational library is the BCL.

Table 1-3. *CTS System.Object (Tool-Managed Object Type)*

BCL Types		**CTS Types**
C++/CLI	**C# Programming Language**	**CIL**
`System::Object` (same root-managed object type)	`System.Object` (same root-managed object type)	`object` (same root-managed object type)

This root object type is the `System.Object` reference type. When you declare a variable of the `object` type (CTS model definition)/`System.Object` (BCL) reference type using any high-level programming language like C#, C++/CLI, F#, VB.NET, and so on, the compiler generates intermediate code using the `object` keyword of the CIL. Table 1-4 summarizes this sequence in a straightforward way.

Table 1-4. *Contextual Resources and Their Fundamental Purposes*

.NET Framework Specialized Applications	Applications, Services, Components, Libraries, and Frameworks
.NET Framework	The software development kit (SDK) includes specialized tools for software development, analysis, deployment, and some management. Specialized components, libraries, and frameworks.
CLR	Implementation of specialized managed environment based on the CLI specification. Uses the resources of the underlying hardware and operating system platform, like Microsoft Windows. Adaptable and capable of using the specialized resources of the underlying hardware and operating system, like Microsoft Windows 10 and Microsoft Windows Server 2016, for example.

About the Virtual Execution System

The VES provides an environment for running managed code, and including features such as security boundaries and automatic memory management via a garbage collector mechanism.

More abstractly, the VES is also known as an *execution engine* or *execution environment*. This execution system is responsible for loading, instantiating, executing, and keeping the cohesiveness of the interactions between the instances, in summary it offers support to the entire lifecycle of the type instances.

As we learned, this book will help us understand the concepts, architecture, and details of the implementation of these two fundamental areas of the platform: the CTS and the VES. Two fundamental built-in types, `string` and `array`, are used as a starting point to help us understand various aspects of the CTS and VES. These built-in platform types are present in any kind of software, so they stand to orthogonal elements. But the .NET Framework also has the BCL special foundational library, part of the CLI specification, that supplies the specialized types necessary to design and implement any kind of software. We'll use the `System.Object`, `System.String`, and `System.Array` reference types as a starting point and deconstruct many parts of their implementations, like some of the interface types that are implemented by these types and how many aspects of these types are used with specialized frameworks such as Windows Forms,

Windows Presentation Foundation (WPF), Universal Windows Platform (UWP) applications, and ASP.NET. The VES provides direct support for a set of built-in platform types, defines a hypothetical machine with an associated machine model and state, has a set of control flow constructs, and offers an exception handling model. To a considerable extent, the purpose of the VES is to provide the support required to execute the CIL instruction set, and is the system that implements and enforces the CTS model. For example, the VES is responsible for loading and running programs written to the managed environment and provides the services needed to execute managed code and work with data, and uses metadata for, at runtime, connect separately generated modules. The VES is also known as the *execution engine*.

Why is that?

Well, it's necessary to start with a few more questions to provide the answer.

About the Module

When you write code using C++, the result of the compiled and linked code is a binary file in a specific format. In this case, we are working with Portable Executable/Common Object File Format (PE/COFF), which is used by the Microsoft Windows operating system. When you write code using C# or C++/CLI, or any other programming language or group of extensions adhering to the CLI specification, the resulting binary file is in the same PE/COFF format, but with some data structures changed or included to support the requirements described by the CLI specification and aspects of the Microsoft Windows operating system way of work.

Currently, on Microsoft Windows, this CLI PE/COFF module can have an `.exe`, `.dll`, `.netmodule`, or `.WinMD` extension. In fact, the existence of an extension is not a requirement, but it's a good practice and accepted standard. If you are using .NET Core (don't confuse this with the .NET Framework) on different operating system/hardware platforms, the extensions and file formats are specific to software and hardware environments, but the fundamental structural resources defined in the CLI as a starting point are the same.

One of the responsibilities of the VES is to load the CLI PE/COFF modules. Part of this loading step includes verifying some structural rules about the file format and guaranteeing that all the information is as expected. The VES uses the metadata information in the CLI PE/COFF modules to verify whether the structural aspects are recognized by the rules and to indicate whether they are valid, required, or optional. If the structural elements exist and are valid, the next step is applying the rules based

on the nature of the elements and the context of use. For example, if the element is a managed type, the execution system needs to verify whether it is a value type or a reference type. If the element is an assembly reference type, one of the responsibilities of this type describes various characteristics of the managed module (structural and behavioral), such as the relationships to other managed modules, what managed types are in it, and what is in any other managed module.

About the Assembly

This is one of the most common questions. As defined and described by the CLI, an *assembly* is a logical unit for the management and deployment of resources designed to work together. In an implementation of the CLR, assemblies can be static or dynamic.

About the Static Assembly

Static assemblies are those stored in a storage device, like a typical hard disk. On the Microsoft Windows operating system, the file format of each module is the CLI PE/COFF. These assemblies have typical .NET Framework types and other specialized resources (audio/video files, localization support files, images, and custom files created specifically to the application), depending of the purpose of each application. The assemblies and modules that are part of the implementation of the .NET Framework and .NET Core are examples, as listed here:

- Assembly `mscorlib`
  - Module `mscorlib.dll`
  - Module `System.Runtime.dll`
  - Module `netstandard.dll`
- Assembly `System.Activities` (part of Microsoft Windows Workflow Foundation)
  - Module `System.Activities.dll`
- Assembly `System.Diagnostics.Debug`
  - Module `System.Diagnostics.Debug.dll`
  - Module `System.dll`
  - Module `netstandard.dll`

About the Dynamic Assembly

Dynamic assemblies are created dynamically at runtime and are created via a specialized application programming interface (API) of the .NET Framework/.NET Core. These dynamic assemblies are created and executed directly in memory. But can the dynamic assembly be saved on a storage device? Yes, but only after being executed.

In a typical project, we have many files, binaries with executable code, and binaries for other types of data such as images but that are part of the software. How do we describe, verify, and enforce the relations and dependencies among them?

The *metadata* has part of the responsibility of assigning available resources to do these tasks.

Working with Assemblies and Modules

A static assembly or a dynamic assembly is a way of keeping the cohesiveness of the types and resources that are designed to work together.

Deployment, Execution and Management

The information stored in the modules and created through assemblies is what helps the runtime environment understand and apply the rules to the relations among the elements.

Let's use a typical *static assembly* in an example. There are four elements of a static assembly.

- CIL that implements all the types and required logic to the module
- Metadata
- The resources (audio/video files, localization support files, images, and custom files created specifically for the application)
- The assembly manifest

From the perspective of the runtime environment and basic structural rules described in the CLI, of these four elements, only the assembly manifest is a required item. But, considering even the simplest application or component, if you don't have the other elements, the application or component doesn't have a practical use, except to learn about the assemblies and modules.

Structural Organization of Elements in a Module

We will start with a basic example here and continue with more details in Chapter 2.

Follow these steps:

1. Using the code editor of your preference, create a simple file and save it with the name `Example00.il` in a directory of your choice that can be used to build source code.
2. Open (as an administrator) one of the developer command prompts installed and configured by Microsoft Visual Studio 2017.
3. Copy this sequence (only two lines of code) of CIL code into the file `Example00.il` and save the file:

```
.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch01 {}
```

4. At the developer command prompt, write the following command:

```
ilasm /DLL Example00.il
```

5. If the code compiles without error, as output we have a binary file with the name `Example00.DLL`.

Following these steps, we have created a *single-file static assembly*, with only the assembly manifest.

Using the ILDasm Tool

With the code compiled correctly and the binary generated, we now can use the ILDasm tool. On the same command prompt that we used to compile code, write the following command:

```
ildasm Example00.DLL
```

ILDasm stands for "intermediate language disassembler."

With the module `Example00.DLL` loaded by the `ILdasm.exe` tool, we will see a screen like Figure 1-1.

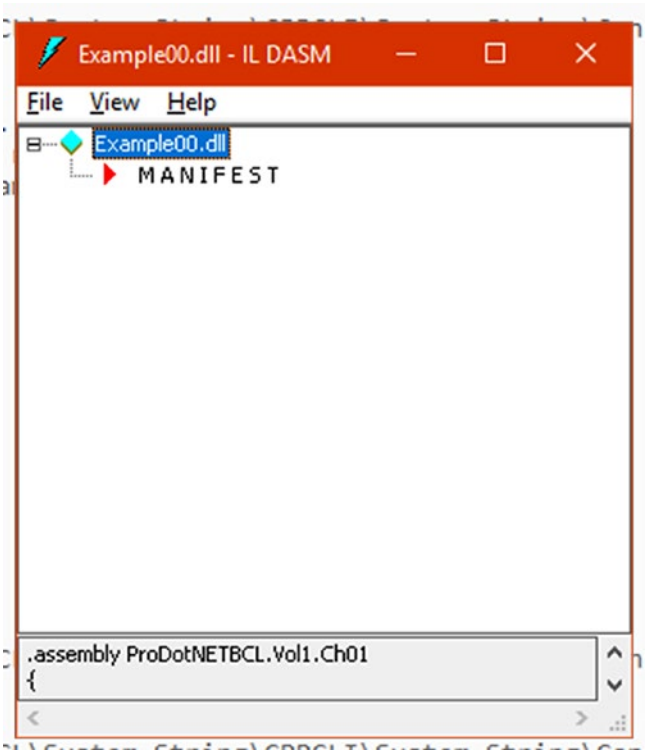


Figure 1-1. *ILDasm showing a single-file static assembly*

Now, double-click in the manifest; a new window opens with information about the assembly manifest (Figure 1-2).

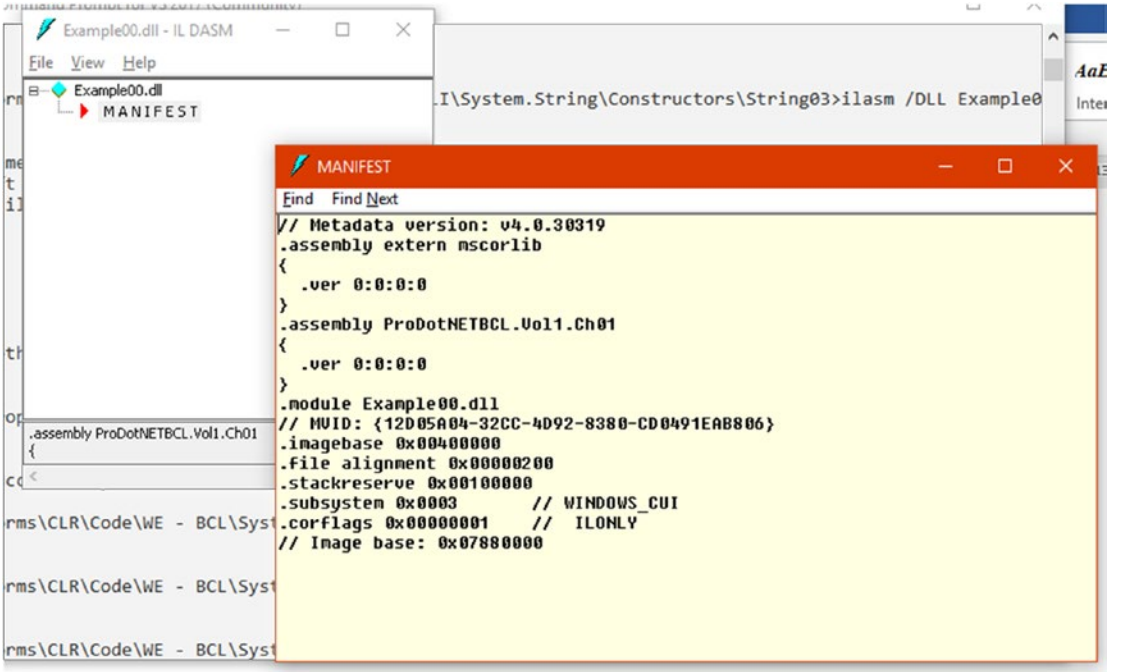


Figure 1-2. *ILDasm showing the assembly manifest of a single-file static assembly*

Implementing the Entry-Point Method

As mentioned we have created a single-file static assembly, with only the assembly manifest. If we want to create an EXE file, we need to change the source code. Using the same `Example00.il` file, update the source code to include a managed method that is the entry point, as shown here:

```
.assembly extern mscorlib {}
.assembly ProDotNETBCL.Vol1.Ch01 {}

.method static public void MyEntryPointMethod() cil managed {

.entrypoint

        ret
}
```

As we can see, the name of the entry-point method doesn't need to be `main`.

To build this code, use the following command:

```
ilasm Example00.il
```

After compiling without error and with the binary generated, we can use the `ILdasm.exe` tool to load the module `Example00.EXE`. We also have the assembly manifest, as shown in Figure 1-3.

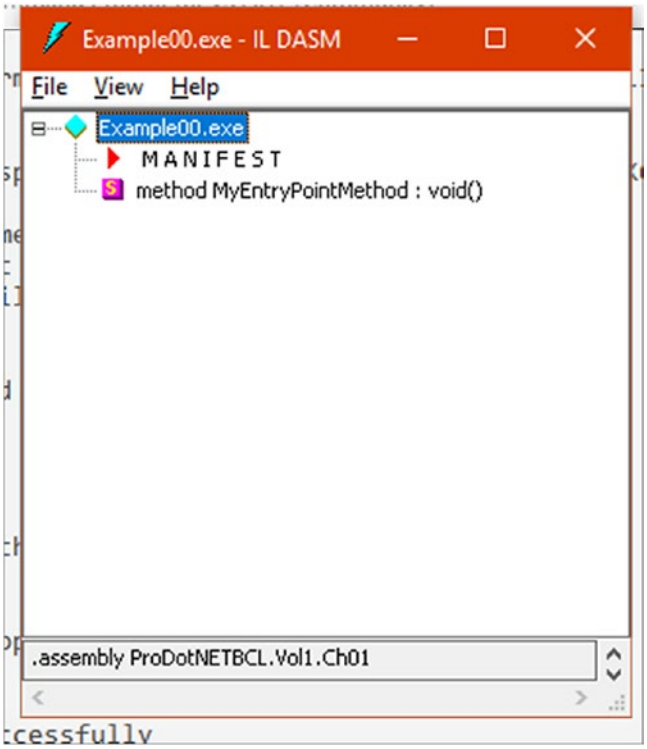


Figure 1-3. *ILDasm showing a single-file static assembly*

We have created a single-file static assembly with an assembly manifest and one method (in this case, the entry-point method). `Example00.EXE` runs like any other .NET managed executable (Figure 1-4).

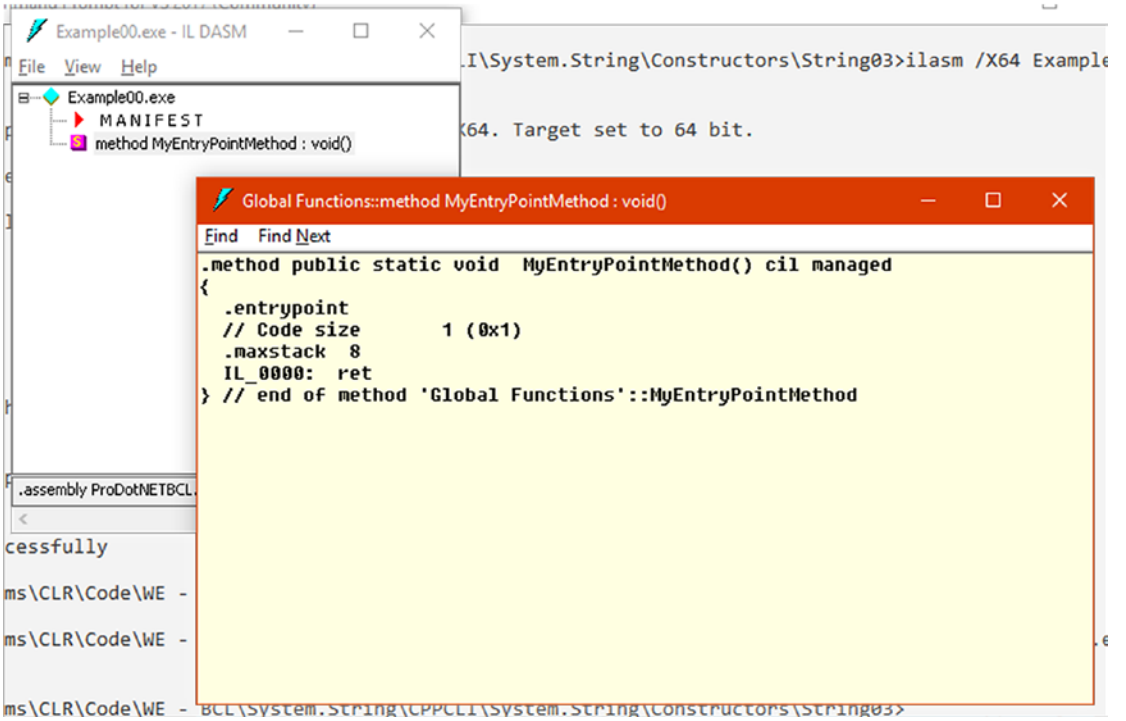


Figure 1-4. *ILDasm showing a single-file static assembly with an assembly manifest and one managed method*

Listing 1-5 shows an example of managed instructions from one of the sample projects including in the companion content of the book. The `.module` directive indicates the name of the binary module. In this case, it is `System.String.String03.EXE`. The `.assembly` directive describes which assemblies make this a logical unit of management and deployment of resources designed to work together. The `.assembly System.String.String03` directive (without the `extern` keyword) describes that this assembly is in the current module. Using the `.assembly extern` directive describes the assembly and where the types are that your `.assembly` / `.module` is referencing. For example, `.assembly extern mscorlib` indicates that the assembly `System.String.String03` is using one or more types of the assembly `mscorlib`. The highlighted CIL instructions are the same that you see in the `Example00.DLL`/`Example00.EXE` modules. We'll talk more about these and other instructions in Chapter 2 and gradually explain more details throughout the book.

Listing 1-5. Fundamental Keywords Used by Static Assemblies or Dynamic Assemblies

```
// Metadata version: v4.0.30319
.module System.String.String03.exe
.assembly System.String.String03
{

.custom instance void [mscorlib]System.CLSCompliantAttribute::.ctor(bool) =
( 01 00 01 00 00 )
.permissionset reqmin= {[mscorlib]System.Security.Permissions.
SecurityPermissionAttribute = {property bool 'SkipVerification' =
bool(true)}}
.hash algorithm 0x00008004

.ver 1:0:0:0

}
.assembly extern mscorlib {
.publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
.hash = (A2 20 0B E6 3F 2B 59 04 0C 46 0B 66 63 01 DF 23
7D 56 8F F8 )
.ver 4:0:0:0
}
.assembly extern System
{
.publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
.hash = (92 DD 47 08 2E 4E E4 27 71 52 40 D9 9B 63 B2 CB
B4 4F 48 D2 )
.ver 4:0:0:0}
```

As we can see, the VES handles a lot of work. But there are more interesting functionalities within this mechanism.

Chapter 2 will cover details about these resources and the CTS and VES. That is, it will cover more about fundamental built-in types and about how the execution environment deals with these types and structural elements of the platform. Initially, we use code written directly in CIL to explain more details about the use of the types and to better understand how to work with modules and assemblies. Then we will use some code in C++ to understand some internal aspects of the execution environment and some special types. From that point, we'll start our journey about the foundational BCL using C++/CLI and the C# programming language.

CHAPTER 2

The Base Class Library

This chapter will cover the BCL foundational library.

The Importance of the BCL

.NET has an extensive set of standard libraries; they are organized into two groups: the core set, which is the BCL, and the complete set, which is the FCL. Both the core set and the complete set are not static (closed); in other words, they are continually being improved by Microsoft and, in the case of the core set for the .NET Core and Mono implementations, by community contributions. The BCL is expected to be part of any .NET implementation as stated by the CLI specification. Because of this role, the BCL provides the most foundational standards and resources upon which other .NET libraries are designed.

As we already know, the BCL and other fundamental libraries are part of the CLI specification, which organizes types and functionalities into *libraries*. These libraries have *profiles* that group the types and functionalities. The profiles are not represented in the metadata; they are logical organizations to facilitate the composition of the libraries and to describe which are the required types and functionalities and which are the optional types and functionalities in some scenarios. The most fundamental required types and functionalities are part of the Kernel profile, and there are two libraries in this profile: the BCL and the Runtime Infrastructure Library. All conforming CLI implementations must support these two libraries.

An Array from the CTS and VES Perspective

In Chapter 1, we learned about two fundamental built-in types, `array` and `string`. All arrays derive from the `System.Array` abstract reference type, which is part of the CTS and directly supported by the VES. The VES is described and supported in the metadata and

R. Villela, *Pro .NET Framework with the Base Class Library*, https://doi.org/10.1007/978-1-4842-4191-2_2

is part of the CLS. To the `System.Array` abstract reference type, this means that the array is not part of any external library in particular; instead, it is an element present in any CLR implementation.

The Importance of the Standardization

The .NET Standard provides a formal specification for .NET APIs that will be available on all implementations of .NET.

The .NET Standard is an API specification common to all implementations of the .NET platform, such as the .NET Framework, .NET Core, and Mono. So, the fundamental purpose of the .NET Standard is to provide a uniform standard for the BCL, with a focus on portability. Beyond that, the .NET Standard is designed to be used as a framework, because our projects can choose different versions of the .NET Standard as the target framework.

The objective of any standard is to provide access to the technologies to as large an audience as possible, and the .NET Standard is no different. When we are creating .NET libraries and they are designed to work with more than one platform, we should consider using the .NET Standard.

Based on these characteristics, development teams and independent developers can create common libraries that are usable in different implementations of .NET. These are called portable libraries.

One of these characteristics is the uniformization of the behaviors and fundamental data structures. In the BCL there is one root reference type: `System.Object`. All reference types and value types derive directly or indirectly from the root `System.Object` reference type. All managed types carry some standard behaviors that cannot be changed, except that we create a specialized implementation of the CLR. These standard behaviors are designed and described in the CTS, implemented and enforced by the VES, and represented via metadata. In the reference types and value types, some of these standard behaviors are expressed in the form of methods.

Let's start with a really fundamental operation.

The Equality Operation

The equality standard behavior is one of the most well-known operations. For any reference type, there is this rule:

They are equal if two references point to the same instance.

For any value type, there is the following rule:

They are equal if two values, even of distinct value types, have the same value.

The standard behavior for the equality operation is part of the VES implementation. If we create two instances of any reference type without any specialized implementation of the equality operation, all reference types are compared using the previous rule for reference types. We do not have to write a single line of code to get this standard behavior; it is part of the supported characteristics implicitly available for any reference type. Listing 2-1 and Listing 2-2 show an example of this.

Listing 2-1. Equality Operation to Reference Type

```
/* C++/CLI (Common Language Infrastructure). */

Object^ objA{ gcnew Object() };
Object^ objB{ gcnew Object() };

Console::WriteLine( "objA == objB: {0}", ( objA == objB ).ToString() );

/* C# programming language. */

object objA = new object();
object objB = new object();

Console.WriteLine( "objA == objB: {0}", ( objA == objB ).ToString() );
```

Listing 2-2. Equality Operation to Value Type

```
/* C++/CLI (Common Language Infrastructure). */

Int32 vltA{ 100 };
Int32 vltB{ 200 };

Boolean isEqual{ vltA == vltB };

Console::WriteLine( "vltA == vltB: {0}", isEqual.ToString() );
```

```
/* C# programming language. */

int vltA = 100;
int vltB = 200;

bool isEqual = ( vltA == vltB );

Console.WriteLine( "vltA == vltB: {0}", isEqual.ToString() );
```

Let's understand more about the equality operation using one special type. The CTS defines the fundamental built-in reference type `System.Array` as a special type.

Special Types

Special types are those that are referenced from the CIL, but no definition is supplied in any assembly; it is the VES that automatically supplies the definitions based on the available information from the reference. It is important to not confuse these special types with the types supplied through the BCL.

`System.Array` is a special type in the VES. How is that?

In the CLI/CTS, arrays are classified as *vector arrays* and *nonvector arrays*. From that distinction comes the logic that is part of the VES implementation, described here:

- Every instance derivation of `System.Array` with a single-dimensional array with zero lower bound is classified as a *vector array* in the CLI. When created, it has a fixed size and element type. We must remember that after it's created, we cannot change an instance of the array (do not confuse this with the elements on the array). For example, we cannot alter the size of the instance (the number of elements). The only way to change the size is by creating a new instance with the requested new size and copying the values of interest.
- When created, a multidimensional array (known as a *nonvector array*) also has as the base class the `System.Array` (BCL) abstract reference type. The multidimensional array can have more than one dimension, and the lower bounds are different from zero to each dimension. Only the VES can create a derivation from the `System.Array` reference type. When created, a derivation is based on the information expressed through the syntax of the programming language used.

The CIL instruction set has special instructions for work with vector arrays. That is, when we declare a single-dimensional array using the C++/CLI syntax or C# programming language syntax, we have a vector array. In fact, this is true of every language to the CLR because the managed platform is language-agnostic.

In Listing 2-3, we have used C++/CLI extensions in the declaration and instantiation of an array. In Listing 2-4, we have used C# programming language in the declaration and instantiation of an array. In Listing 2-5, we have used the CIL programming language in the declaration and instantiation of an array, and `newarr`, `stelem.i4`, `ldlen`, and `ldelem.i4` are examples of these specialized intermediate language instructions.

Listing 2-3. Using C++/CLI to Create a Single-Dimensional Array

```
/*
Simple definition of Person custom reference type.
This is used to this simple code compile.
*/
public ref class Person {

public:
      Person( String^ name ) { return; };

};

::cli::array<Person^>^ people = { gcnew Person( "A" ), gcnew Person( "B" ),
gcnew Person( "C" ) };

::cli::array<Int32>^ numbers = { 0i32, 1i32, 2i32, 3i32 };
```

Listing 2-4. Using C# to Create a Single-Dimensional Array

```
// Simple definition of Person custom reference type.
public class Person {
       public Person( String name ) { return; };

};

Person[] peopleOne = { new Person( "A" ), new Person( "B" ), new Person( "C" ) };
```

```
Person[] peopleTwo = new Person[] { new Person( "A" ), new Person( "B" ),
new Person( "C" ) };
Person[] peopleThree = new Person[ 3 ] { new Person( "A" ), new Person( "B" ),
new Person( "C" ) };
Int32[] numbers = { 0, 1, 2, 3 };
```

Listing 2-5. Using the CIL to Create a Single-Dimensional Array

```
/*

stelem.i4 (Store a value in a vector element of type int32) with the
operation code (opcode) value 0x9E.

*/

.assembly extern mscorlib {}

.assembly ProDotNETBCL.Ch02 {
}

.method static public void MyEntryPointMethod() cil managed {

.entrypoint

/* Declare the vector and initialize the associated allocated block of
managed memory . */

.locals init ( int32[] numbers, int32 index, int32 item )

/* Load constant with the number of items for the new vector. */
ldc.i4.s            0x000A

/* Vector creation. */
newarr [mscorlib] System.Int32

stloc.0
call   void [mscorlib]System.Console::Clear()
ldstr "\nVector numbers created.\n"
call   void [mscorlib]System.Console::WriteLine(string)

ldloc.0
```

```
/* Third position. */
ldc.i4.s            0x0002
stloc.1
ldloc.1

/* Value to store. */
ldc.i4.s            0x72
stloc.2
ldloc.2

/* Element storing. */
stelem.i4

ldloc.0

ldlen

stloc.1
ldstr "\nVector length: {0}.\n"
ldloca.s index
call instance string [mscorlib] System.Int32::ToString()
call   void [mscorlib]System.Console::WriteLine(string, object)

ldstr "\nVector item {0} at index {1}.\n"

ldloc.0

/* Third position. */
ldc.i4.s 0x0002

/* Element loading. */
ldelem.i4

stloc.2
ldloca.s item
call instance string [mscorlib] System.Int32::ToString()

/* Third position. */
ldc.i4.s 0x0002
stloc.1
ldloca.s index
```

```
call instance string [mscorlib] System.Int32::ToString()

call   void [mscorlib]System.Console::WriteLine(string, object, object)

ldstr "\n\n\nPress <ENTER> to finish..."
call   void [mscorlib]System.Console::WriteLine(object)
call   string [mscorlib]System.Console::ReadLine()
pop

ret

}
```

To create and use multidimensional arrays, we must follow the syntax provided by each programming language. Listing 2-6 and Listing 2-7 show examples of this using C++/CLI and C#, respectively.

Listing 2-6. Creating an Instance of a Multidimensional Array with C++/CLI

```
constexpr Int32 dimensions { 2i32 };
::cli::array<String^, dimensions>^ list = gcnew array<String^,
dimensions>( 10i32, 6i32 );
```

Listing 2-7. Creating an Instance of a Multidimensional Array with C#

```
Int32[,] myArray = new Int32[ 10, 6 ];
```

As described in the Microsoft official documentation, `System.Array` does not provide constructors that we can use from client code; however, the documentation does say that we can use the `System.Array.CreateInstance()` static method as a way to create an instance of arrays. However, we should avoid the use of this method and use the syntax provided by the programming language. The reason is pretty simple; if we use the `Array.CreateInstance()` static method, we are clearly saying to the compiler that we are using late-bound. When using the `Array.CreateInstance()` static method, the code becomes more complex and doesn't offer any advantages. It is more difficult for the compiler to do some optimization, because late-binding means that many verifications and actions are made only at runtime and not at compile time. For example, when we declare a single-dimensional array using the specialized syntax of each programming language, the respective programming language compiler can choose the most adequate

sequence of CIL instructions to be emitted. When we use the `Array.CreateInstance()` static method, the compiler cannot do this kind of optimization because we deliberately delayed these actions to be realized at runtime. We must be aware that there is nothing wrong with the `Array.CreateInstance()` method and other methods and managed types that use late-binding features, but to most of the code used in business applications, there is no relevant gain.

Listing 2-8 and Listing 2-9 show examples of the `Array.CreateInstance()` static method using C++/CLI and C#, respectively.

Listing 2-8. Using the Array::CreateInstance() Static Method in C++/CLI

```
/* This is not the recommended way to create instances of arrays. */

Array^ otherArray{ Array::CreateInstance( Int32::typeid, 10i32 ) };
```

Listing 2-9. Using the Array.CreateInstance() Static Method in C#

```
// This is not the recommended way to create instances of arrays.

Array otherArray = Array.CreateInstance( typeof( int ), 10 );
```

In Listing 2-10, we have the generated intermediate language code, where we can see that the instructions are the same as used to declare any other typical reference type. There are no special CIL instructions to deal with multidimensional arrays (nonvector arrays).

Listing 2-10. Intermediate Code Using the System.Array Abstract Reference Type

```
.locals init ( class [mscorlib]System.Array otherArray )

ldc.i4.s   10
call       class [mscorlib]System.Array [mscorlib]System.
           Array::CreateInstance(class [mscorlib]System.Type, int32)
```

But when we try to do certain operations using arrays created via `Array.CreateInstance()`, like a simple assignment of `moreObjects[ indexOne, indexTwo ] = new Object()` or `moreObjects[ index ] = new Object()`, we cannot do it. To do operations with arrays created with `Array.CreateInstance()`, we must use specialized static and instance methods of the `System.Array` abstract reference type. To set a value,

we must use the `Array.SetValue()` instance method. To get a value, we must use the `Array.GetValue()` instance method. As we can read in the generated intermediate language documentation, there is a box CIL instruction because the `Array.SetValue()` and `Array.GetValue()` instance methods are not implemented using generics. Therefore, the first parameter of the set operation is of the `System.Object` reference type, and when we pass a value type as an argument, the box operation is used. The return type of the get operation is of `System.Object`, and if the base type of the elements of the array is a value type, we will use the unboxing operation on the code (explicitly via casting or implicitly via compiler-generated code). In Listing 2-11, Listing 2-12, and Listing 2-13, and Listing 2-14 we can read about the use of these two instance methods, `Array.SetValue()` and `Array.GetValue()`.

Listing 2-11. Using Instance Methods Array ➤ SetValue and Array ➤ GetValue (Example in C++/CLI)

```
/* This is not the recommended way to create instances of arrays. */

Array^ otherArray{ Array::CreateInstance( Int32::typeid, 10i32 ) };

Int32 index{};
Int32 length{ otherArray->Length };

for (; index != length; index++ ) otherArray->SetValue( index, index );

for ( index = {}; index != length; index++ )
 Console::WriteLine( "Value: {0} at index {1}\n", otherArray->GetValue( index ),
 index.ToString() );
```

Listing 2-12. Using Instance Methods Array.SetValue() and Array.GetValue() (Example in the CIL from the C++/CLI Code of Listing 2-11)

```
.locals ([0] int32 index,
         [1] class [mscorlib]System.Array otherArray,
         [2] int32 length,
         [3] int32 V_3,
         [4] int32 V_4)
```

```
IL_0000:  ldc.i4.0
IL_0001:  stloc.3
IL_0002:  ldnull
IL_0003:  stloc.1
IL_0004:  ldtoken    [mscorlib]System.Int32
IL_0009:  call       class [mscorlib]System.Type [mscorlib]System.
                     Type::GetTypeFromHandle(valuetype [mscorlib]System.
                     RuntimeTypeHandle)
IL_000e:  ldc.i4.s   10
IL_0010:  call       class [mscorlib]System.Array [mscorlib]System.Array::
                     CreateInstance(class [mscorlib]System.Type, int32)
IL_0015:  stloc.1
IL_0016:  ldc.i4.0
IL_0017:  stloc.0
IL_0018:  ldloc.1
IL_0019:  call       instance int32 [mscorlib]System.Array::get_Length()
IL_001e:  stloc.2
IL_001f:  br.s       IL_0025
IL_0021:  ldloc.0
IL_0022:  ldc.i4.1
IL_0023:  add
IL_0024:  stloc.0
IL_0025:  ldloc.0
IL_0026:  ldloc.2
IL_0027:  beq.s      IL_0038
IL_0029:  ldloc.1
IL_002a:  ldloc.0
IL_002b:  box        [mscorlib]System.Int32
IL_0030:  ldloc.0
IL_0031:  call       instance void [mscorlib]System.Array::SetValue(object,
                     int32)
IL_0036:  br.s       IL_0021
IL_0038:  ldc.i4.0
IL_0039:  stloc.0
```

```
IL_003a:  br.s       IL_0040
IL_003c:  ldloc.0
IL_003d:  ldc.i4.1
IL_003e:  add
IL_003f:  stloc.0
IL_0040:  ldloc.0
IL_0041:  ldloc.2
IL_0042:  beq.s      IL_0061
IL_0044:  ldloc.0
IL_0045:  stloc.s    V_4
IL_0047:  ldstr      "Value: {0} at index {1}\n"
IL_004c:  ldloc.1
IL_004d:  ldloc.0
IL_004e:  call       instance object [mscorlib]System.Array::GetValue(int32)
IL_0053:  ldloca.s   V_4
IL_0055:  call       instance string [mscorlib]System.Int32::ToString()
IL_005a:  call       void [mscorlib]System.Console::WriteLine(string,
                                                              object,
                                                              object)
IL_005f:  br.s       IL_003c
```

Listing 2-13. Using Instance Methods Array.SetValue() and Array.GetValue() (Example in C#)

```
// This is not the recommended way to create instances of single-dimensional
or multidimensional arrays.

Array otherArray = Array.CreateInstance( typeof( Int32 ), 10 );
Int32 index = new Int32();
Int32 length = otherArray.Length;
for ( ; index != length; index++ ) otherArray.SetValue( index, index );
for ( index = new Int32(); index != length; index++ )
Console.WriteLine( "Value: {0} at index {1}\n", otherArray.GetValue( index ),
index.ToString() );
```

Listing 2-14. Using Instance Methods Array.SetValue() and Array.GetValue() (Example in the CIL from the C# Code of Listing 2-13)

```
.locals init ([0] class [mscorlib]System.Array otherArray,
              [1] int32 index,
              [2] int32 length,
              [3] bool V_3,
              [4] bool V_4)

IL_0000:  nop
IL_0001:  ldtoken    [mscorlib]System.Int32
IL_0006:  call       class [mscorlib]System.Type [mscorlib]System.
                     Type::GetTypeFromHandle(valuetype [mscorlib]System.
                     RuntimeTypeHandle)
IL_000b:  ldc.i4.s   10
IL_000d:  call       class [mscorlib]System.Array [mscorlib]System.
                     Array::CreateInstance(class [mscorlib]System.Type,
                     int32)
IL_0012:  stloc.0
IL_0013:  ldc.i4.0
IL_0014:  stloc.1
IL_0015:  ldloc.0
IL_0016:  callvirt   instance int32 [mscorlib]System.Array::get_Length()
IL_001b:  stloc.2
IL_001c:  br.s       IL_0030
IL_001e:  ldloc.0
IL_001f:  ldloc.1
IL_0020:  box        [mscorlib]System.Int32
IL_0025:  ldloc.1
IL_0026:  callvirt   instance void [mscorlib]System.Array::SetValue(object,
                     int32)
IL_002b:  nop
IL_002c:  ldloc.1
IL_002d:  ldc.i4.1
IL_002e:  add
IL_002f:  stloc.1
IL_0030:  ldloc.1
```

```
IL_0031:  ldloc.2
IL_0032:  ceq
IL_0034:  ldc.i4.0
IL_0035:  ceq
IL_0037:  stloc.3
IL_0038:  ldloc.3
IL_0039:  brtrue.s   IL_001e
IL_003b:  ldc.i4.0
IL_003c:  stloc.1
IL_003d:  br.s       IL_005c
IL_003f:  ldstr      "Value: {0} at index {1}\n"
IL_0044:  ldloc.0
IL_0045:  ldloc.1
IL_0046:  callvirt   instance object [mscorlib]System.Array::GetValue(int32)
IL_004b:  ldloca.s   index
IL_004d:  call       instance string [mscorlib]System.Int32::ToString()
IL_0052:  call       void [mscorlib]System.Console::WriteLine(string,
                                                              object,
                                                              object)
IL_0057:  nop
IL_0058:  ldloc.1
IL_0059:  ldc.i4.1
IL_005a:  add
IL_005b:  stloc.1
IL_005c:  ldloc.1
IL_005d:  ldloc.2
IL_005e:  ceq
IL_0060:  ldc.i4.0
IL_0061:  ceq
IL_0063:  stloc.s    V_4
IL_0065:  ldloc.s    V_4
IL_0067:  brtrue.s   IL_003f
```

The Equality Operation and Vector Arrays and Nonvector Arrays

The following two listings show the scenarios for an equality operation on single-dimensional arrays. First, Listing 2-15 is implemented in C++/CLI, and Listing 2-16 is implemented using C#. The single-dimensional array called `morePeople` reference type receives a copy of the handle to the object of the single-dimensional array `people`. Because of this, the standard equality operation returns true.

Listing 2-15. Equality Operation with Single-Dimensional Arrays (Example in C++/CLI)

```
/*
Simple definition of Person custom reference type.
This is used to this simple code compile.
*/
public ref class Person {

public:
      Person( String^ name ) { return; };

};

::cli::array<Person^>^ people = { gcnew Person( "A" ), gcnew Person( "B" ),
gcnew Person( "C" ) };

::cli::array<Person^>^ morePeople = people;

Console::WriteLine( "people == morePeople: {0}", ( people == morePeople ).
ToString() );
```

Listing 2-16. Equality Operation with Single-Dimensional Arrays (Example in C#)

```
// Simple definition of Person custom reference type.
public class Person {
       public Person( String name ) { return; }
};
```

```
Person[] people = { new Person( "A" ), new Person( "B" ), new Person( "C" ) };
Person[] morePeople = people;

Console.WriteLine( "people == morePeople: {0}", ( people == morePeople ).
ToString() );
```

In the Listing 2-17 implementation in C++/CLI and in the Listing 2-18 implementation in C#, the single-dimensional array objects and single-dimensional array `moreObjects` are declared with the same base type, `System.Object`. However, each element of the arrays has a different base type, but each is identical between instances, including the position (index); they also have the same values but are distinct instances. The result of the standard equality operation returns false because the array objects and the array `moreObjects` point to different instances of the array. As shown in Listing 2-17, which is implemented in C++/CLI, and in Listing 2-18, which is implemented in C#, the instances of arrays, being reference types, are compared by the standard equality behavior to reference types.

Listing 2-17. Equality Operation with Vector Arrays (Example in C++/CLI)

```
/*
Simple definition of Person custom reference type.
This is used to this simple code compile.
*/
public ref class Person {

public:
      Person( String^ name ) { return; };

};


      ::cli::array<Object^>^ objects = { gcnew Person( "A" ),
      gcnew String( "C++/CLI" ), 10ui32 };

      ::cli::array<Object^>^ moreObjects = { gcnew Person( "A" ), gcnew
      String( "C++/CLI" ), 10ui32 };

      Console::WriteLine( "objects == moreObjects: {0}", ( objects ==
      moreObjects ).ToString() );
```

Listing 2-18. Equality Operation with Vector Arrays (Example in C#)

```
// Simple definition of Person custom reference type.
public class Person {
       public Person( String name ) { return; }
};

Object[] objects = { new Person( "A" ), "C# Programming Language", 10 };
Object[] moreObjects = { new Person( "A" ), "C# Programming Language", 10 };

Console.WriteLine( "objects == moreObjects: {0}", ( objects ==
moreObjects ).ToString() );
```

The multidimensional (nonvector) arrays are reference types too, and the examples in the Listing 2-19 implementation in C++/CLI and the Listing 2-21 implementation in C# show the behavior of the equality operation. The Listing 2-20 shows the CIL generated from C++/CLI implementation and the Listing 2-22 shows the CIL generated from the C# implementation.

First, in Listing 2-19, the reference type of the multidimensional array called `morePeople` receives a copy of the handle to the object of the multidimensional array `people`, and therefore the standard equality operation returns true.

Listing 2-19. Equality Operation with Nonvector Arrays (Example in C++/CLI)

```
/*

Simple definition of Person custom reference type.

*/

public ref class Person {
public:
       Person( String^ name ) { return; };
};

constexpr Int32 dimensions{ 2i32 };
```

```
array<Person^, dimensions >^ people = gcnew array<Person^, dimensions >{
            { gcnew Person( "A" ) },
            { gcnew Person( "B" ) },
            { gcnew Person( "C" ) }
};
Array^ morePeople{ people };
Console::WriteLine( "people == morePeople: {0}", ( people ==
morePeople ).ToString() );
```

Listing 2-20. Excerpt of CIL Generated from the C++/CLI Code in Listing 2-19

```
.locals ([0] class Person[0...,0...] V_0,
         [1] class Person[0...,0...] people,
         [2] class [mscorlib]System.Array morePeople,
         [3] int32 V_3,
         [4] bool V_4,
         [5] int32 modopt([mscorlib]System.Runtime.CompilerServices.
             IsConst) dimensions)
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.3
  IL_0002:  ldnull
  IL_0003:  stloc.0
  IL_0004:  ldnull
  IL_0005:  stloc.1
  IL_0006:  ldnull
  IL_0007:  stloc.2
  IL_0008:  ldc.i4.2
  IL_0009:  stloc.s    dimensions
  IL_000b:  ldc.i4.3
  IL_000c:  ldc.i4.1
  IL_000d:  newobj     instance void class Person[0...,0...]::.ctor(int32,
                       int32)
  IL_0012:  stloc.0
  IL_0013:  ldloc.0
  IL_0014:  ldc.i4.0
```

```
IL_0015:  ldc.i4.0
IL_0016:  ldstr      "A"
IL_001b:  newobj     instance void Person::.ctor(string)
IL_0020:  call       instance void class Person[0...,0...]::Set(int32,
                     int32, class Person)
IL_0025:  ldloc.0
IL_0026:  ldc.i4.1
IL_0027:  ldc.i4.0
IL_0028:  ldstr      "B"
IL_002d:  newobj     instance void Person::.ctor(string)
IL_0032:  call       instance void class Person[0...,0...]::Set(int32,
                     int32, class Person)
IL_0037:  ldloc.0
IL_0038:  ldc.i4.2
IL_0039:  ldc.i4.0
IL_003a:  ldstr      "C"
IL_003f:  newobj     instance void Person::.ctor(string)
IL_0044:  call       instance void class Person[0...,0...]::Set(int32,
                     int32, class Person)
IL_0049:  ldloc.0
IL_004a:  stloc.1
IL_004b:  ldloc.1
IL_004c:  stloc.2
IL_004d:  ldloc.1
IL_004e:  ldloc.2
IL_004f:  bne.un.s   IL_0054
IL_0051:  ldc.i4.1
IL_0052:  br.s       IL_0055
IL_0054:  ldc.i4.0
IL_0055:  stloc.s    V_4
IL_0057:  ldstr      "people == morePeople: {0}"
IL_005c:  ldloca.s   V_4
IL_005e:  call       instance string [mscorlib]System.Boolean::ToString()
IL_0063:  call       void [mscorlib]System.Console::WriteLine(string,
                                                              object)
```

```
  IL_0068:  ldloc.1
  IL_0069:  ldc.i4.0
  IL_006a:  ldc.i4.0
  IL_006b:  ldnull
  IL_006c:  call       instance void class Person[0...,0...]::Set(int32,
                       int32, class Person)
```

Listing 2-21. Equality Operation with Nonvector Arrays (Example in C#)

```
// Simple definition of Person custom reference type.
public class Person {
       public Person( String name ) { return; }
};

Person[,] people = new Person[,]{ { new Person( "A" ) },
                  { new Person( "B" ) }, { new Person( "C" ) } };
Array morePeople = people;

Console.WriteLine( "people == morePeople: {0}", ( people ==
morePeople ).ToString() );
```

Listing 2-22. Excerpt of CIL Generated from the C# Code in Listing 2-21

```
.locals init ([0] class RVJ.Listing_2_21.Person[0...,0...] people,
              [1] class [mscorlib]System.Array morePeople,
              [2] bool V_2)
IL_0000:  nop
IL_0001:  ldc.i4.3
IL_0002:  ldc.i4.1
IL_0003:  newobj     instance void class RVJ.Listing_2_21.Person[0...,
                     0...]::.ctor(int32, int32)
IL_0008:  dup
IL_0009:  ldc.i4.0
IL_000a:  ldc.i4.0
```

```
IL_000b:  ldstr      "A"
IL_0010:  newobj     instance void RVJ.Listing_2_21.Person::.ctor(string)
IL_0015:  call       instance void class RVJ.Listing_2_21.Person[0...,0...]::
                                   Set(int32, int32, class RVJ.Listing_2_21.Person)
IL_001a:  dup
IL_001b:  ldc.i4.1
IL_001c:  ldc.i4.0
IL_001d:  ldstr      "B"
IL_0022:  newobj     instance void RVJ.Listing_2_21.Person::.ctor(string)
IL_0027:  call       instance void class RVJ.Listing_2_21.Person[0...,0...]
                                   ::Set(int32, int32, class RVJ.Listing_2_21.
                                   Person)
IL_002c:  dup
IL_002d:  ldc.i4.2
IL_002e:  ldc.i4.0
IL_002f:  ldstr      "C"
IL_0034:  newobj     instance void RVJ.Listing_2_21.Person::.ctor(string)
IL_0039:  call       instance void class RVJ.Listing_2_21.Person[0...,0...]
                                   ::Set(int32, int32, class RVJ.Listing_2_21.
                                   Person)
IL_003e:  stloc.0
IL_003f:  ldloc.0
IL_0040:  stloc.1
IL_0041:  ldstr      "people == morePeople: {0}"
IL_0046:  ldloc.0
IL_0047:  ldloc.1
IL_0048:  ceq
IL_004a:  stloc.2
IL_004b:  ldloca.s   V_2
IL_004d:  call       instance string [mscorlib]System.Boolean::ToString()
IL_0052:  call       void [mscorlib]System.Console::WriteLine(string,
                                                              object)
```

In Listing 2-23, Listing 2-24, Listing 2-25, and Listing 2-26, the multidimensional array objects and multidimensional array `moreObjects` are declared with the same base type, `System.Object`. Each element has a different base type but has identical instances and positions (indices), with the same values; however, they are distinct instances. As shown by these four examples, the nonvector arrays being reference types are compared by the standard equality behavior to the reference types.

Listing 2-23. Equality Operation with Nonvector Arrays (Example in C++/the CLI)

```
/*

Simple definition of Person custom reference type.

*/

public ref class Person {
public:
       Person( String^ name ) { return; };
};

constexpr Int32 dimensions{ 2i32 };

::cli::array<Object^, dimensions> ^ objects = gcnew ::cli::array<Object^,
                                              dimensions>{
          { gcnew Person( "A" ) },
          { gcnew String( "C++/CLI" ) },
          { 10ui32 }
};

Array^ moreObjects{ gcnew array<Object^, dimensions> {
       { gcnew Person( "A" ) },
       { gcnew String( "C++/CLI" ) },
       { 10ui32 }
} };

Console::WriteLine( "objects == moreObjects: {0}", ( objects ==
moreObjects ).ToString() );
```

Listing 2-24. Excerpt of CIL Generated from the C++/CLI Code in Listing 2-23

```
.maxstack 4
.locals ([0] object[0...,0...] V_0,
         [1] object[0...,0...] V_1,
         [2] class [mscorlib]System.Array moreObjects,
         [3] object[0...,0...] objects,
         [4] string V_4,
         [5] bool V_5,
         [6] int32 modopt([mscorlib]System.Runtime.CompilerServices.
             IsConst) dimensions)
IL_0000:  ldnull
IL_0001:  stloc.1
IL_0002:  ldnull
IL_0003:  stloc.3
IL_0004:  ldnull
IL_0005:  stloc.0
IL_0006:  ldnull
IL_0007:  stloc.2
IL_0008:  ldc.i4.2
IL_0009:  stloc.s     dimensions
IL_000b:  ldc.i4.3
IL_000c:  ldc.i4.1
IL_000d:  newobj      instance void object[0...,0...]::.ctor(int32, int32)
IL_0012:  stloc.1
IL_0013:  ldloc.1
IL_0014:  ldc.i4.0
IL_0015:  ldc.i4.0
IL_0016:  ldstr       "A"
IL_001b:  newobj      instance void Person::.ctor(string)
IL_0020:  call        instance void object[0...,0...]::Set(int32, int32,
                                   object)
IL_0025:  ldloc.1
IL_0026:  ldc.i4.1
IL_0027:  ldc.i4.0
IL_0028:  ldstr       "C++/CLI"
```

```
IL_002d:  call       instance void object[0...,0...]::Set(int32, int32,
                                                          object)
IL_0032:  ldloc.1
IL_0033:  ldc.i4.2
IL_0034:  ldc.i4.0
IL_0035:  ldc.i4.s   10
IL_0037:  box        [mscorlib]System.UInt32
IL_003c:  call       instance void object[0...,0...]::Set(int32, int32,
                                                          object)
IL_0041:  ldloc.1
IL_0042:  stloc.3
IL_0043:  ldc.i4.3
IL_0044:  ldc.i4.1
IL_0045:  newobj     instance void object[0...,0...]::.ctor(int32, int32)
IL_004a:  stloc.0
IL_004b:  ldloc.0
IL_004c:  ldc.i4.0
IL_004d:  ldc.i4.0
IL_004e:  ldstr      "A"
IL_0053:  newobj     instance void Person::.ctor(string)
IL_0058:  call       instance void object[0...,0...]::Set(int32, int32,
                                                          object)
IL_005d:  ldloc.0
IL_005e:  ldc.i4.1
IL_005f:  ldc.i4.0
IL_0060:  ldstr      "C++/CLI"
IL_0065:  call       instance void object[0...,0...]::Set(int32, int32,
                                                          object)
IL_006a:  ldloc.0
IL_006b:  ldc.i4.2
IL_006c:  ldc.i4.0
IL_006d:  ldc.i4.s   10
IL_006f:  box        [mscorlib]System.UInt32
IL_0074:  call       instance void object[0...,0...]::Set(int32, int32,
                                                          object)
```

```
IL_0079:  ldloc.0
IL_007a:  stloc.2
IL_007b:  ldloc.3
IL_007c:  ldloc.2
IL_007d:  bne.un.s   IL_0082
IL_007f:  ldc.i4.1
IL_0080:  br.s       IL_0083
IL_0082:  ldc.i4.0
IL_0083:  stloc.s    V_5
IL_0085:  ldloca.s   V_5
IL_0087:  call       instance string [mscorlib]System.Boolean::ToString()
IL_008c:  stloc.s    V_4
IL_008e:  ldstr      "objects == moreObjects: {0}"
IL_0093:  ldloc.s    V_4
IL_0095:  call       void [mscorlib]System.Console::WriteLine(string,
                                                              object)
```

Listing 2-25. Equality Operation with Nonvector Arrays (Example in C#)

```
// Simple definition of Person custom reference type.
public class Person {
       public Person( String name ) { return; }
};

Object[,] objects = new Object[,]{ { new Person( "A" ) }, { "C# programming
                    language" }, { 10 } };
Array moreObjects = new Object[,]{ { new Person( "A" ) }, { "C# programming
                    language" }, { 10 } };

Console.WriteLine( "objects == moreObjects: {0}", ( objects ==
moreObjects ).ToString() );
```

Listing 2-26. Excerpt of CIL Generated from the C# Code in Listing 2-25

```
.maxstack  5
.locals init ([0] object[0...,0...] objects,
              [1] class [mscorlib]System.Array moreObjects,
              [2] bool V_2)
IL_0000:  nop
IL_0001:  ldc.i4.3
IL_0002:  ldc.i4.1
IL_0003:  newobj     instance void object[0...,0...]::.ctor(int32, int32)
IL_0008:  dup
IL_0009:  ldc.i4.0
IL_000a:  ldc.i4.0
IL_000b:  ldstr      "A"
IL_0010:  newobj     instance void RVJ.Listing_2_25.Person::.ctor(string)
IL_0015:  call       instance void object[0...,0...]::Set(int32, int32,
                                  object)
IL_001a:  dup
IL_001b:  ldc.i4.1
IL_001c:  ldc.i4.0
IL_001d:  ldstr      "C# programming language"
IL_0022:  call       instance void object[0...,0...]::Set(int32, int32,
                                  object)
IL_0027:  dup
IL_0028:  ldc.i4.2
IL_0029:  ldc.i4.0
IL_002a:  ldc.i4.s   10
IL_002c:  box        [mscorlib]System.Int32
IL_0031:  call       instance void object[0...,0...]::Set(int32, int32,
                                  object)
IL_0036:  stloc.0
IL_0037:  ldc.i4.3
IL_0038:  ldc.i4.1
IL_0039:  newobj     instance void object[0...,0...]::.ctor(int32, int32)
IL_003e:  dup
```

```
IL_003f:  ldc.i4.0
IL_0040:  ldc.i4.0
IL_0041:  ldstr      "A"
IL_0046:  newobj     instance void RVJ.Listing_2_25.Person::.ctor(string)
IL_004b:  call       instance void object[0...,0...]::Set(int32, int32,
                                   object)
IL_0050:  dup
IL_0051:  ldc.i4.1
IL_0052:  ldc.i4.0
IL_0053:  ldstr      "C# programming language"
IL_0058:  call       instance void object[0...,0...]::Set(int32, int32,
                                   object)
IL_005d:  dup
IL_005e:  ldc.i4.2
IL_005f:  ldc.i4.0
IL_0060:  ldc.i4.s   10
IL_0062:  box        [mscorlib]System.Int32
IL_0067:  call       instance void object[0...,0...]::Set(int32, int32,
                                   object)
IL_006c:  stloc.1
IL_006d:  ldstr      "objects == moreObjects: {0}"
IL_0072:  ldloc.0
IL_0073:  ldloc.1
IL_0074:  ceq
IL_0076:  stloc.2
IL_0077:  ldloca.s   V_2
IL_0079:  call       instance string [mscorlib]System.Boolean::ToString()
IL_007e:  call       void [mscorlib]System.Console::WriteLine(string, object)
```

The point here is that the VES is prepared to understand that there are single-dimensional and multidimensional arrays from the perspective of the CTS, and there are customized behavioral rules, such as being the only component of the CLR capable of explicitly deriving and creating instances of these new derived types.

The Equality Operation and the Fundamental Built-in String Type

The purpose of the string data type is to be a straightforward way to manipulate a sequence of instances of the char data type, another fundamental built-in type in the CTS. For example, in the C++ programming language, if we had to deal with sequences of `char` (1 byte), `wchar_t` (2 bytes), `char16_t` (2 bytes), or `char32_t` (4 bytes) without the support of specialized types like `u16string` from the C++ Standard Library, we would have to use static arrays or dynamically allocated blocks of memory for each one of these data types. If we need to deal with more complex operations, the adequate choice is to use a specialized type from a library like the C++ Standard Library. Listing 2-27 shows a C++ programming language example of manipulating a buffer of characters.

Listing 2-27. Using Buffers for Different Sizes of Character Types (Example in C++)

```
char sequenceOne[]{ u8"C++ programming language." };
wchar_t sequenceTwo[]{ L"C++ programming language." };
char16_t sequenceThree[]{ u"C++ programming language." };
char32_t sequenceFour[]{ U"C++ programming language." };

typedef char16_t BaseType;

constexpr uint32_t BaseTypeSize{ sizeof( BaseType ) };
constexpr uint32_t MaxNumberOfItems{ 100ui32 };
constexpr uint32_t BufferSize{ MaxNumberOfItems << BaseTypeSize };

BaseType* sequence{ ( (BaseType*) calloc( MaxNumberOfItems, BaseTypeSize ) ) };

free( sequence ), sequence = nullptr;


typedef char16_t BaseType;

constexpr uint32_t BaseTypeSize{ sizeof( BaseType ) };
constexpr uint32_t MaxNumberOfItems{ 100ui32 };
constexpr uint32_t BufferSize{ MaxNumberOfItems << BaseTypeSize };

BaseType* sequence{ ( (BaseType*) HeapAlloc( GetProcessHeap(),
HEAP_ZERO_MEMORY, BufferSize ) ) };

HeapFree( GetProcessHeap(), {}, SecureZeroMemory( sequence, BufferSize ) );
```

```
sequence = nullptr;

typedef string STD_BaseType;

STD_BaseType* message{ new STD_BaseType( u8"C++ programming language" ) };

wprintf_s( L"Value: %S\nLength: %u\nEmpty: %s", message->c_str(), message-
>length(),( ( message->empty() ) ? L"Yes" : L"No" ) );

message->clear();

delete message, message = nullptr;
```

As we can see, working with specialized types in specialized operations is a common choice within every context of programming or programming language.

.NET's String Data Type and the Equality Operation

The Unicode standard provides programmers with a single universal character encoding for the different languages, idioms, and dialects adopted by the diverse cultures around the world. Also, the Unicode standard provides data about character functions; provides a way to sort text in those different cultures and their respective languages, idioms, and dialects; and provides other features such as the formatting of numbers, date, and time. The .NET adopted the Unicode standard as one of the fundamental elements of the platform, and the instances of *value type* `System.Char` are used to deal with fundamental aspects of the Unicode standard. In Listing 2-28, we show a declaration of the `System.Char` value type, just to simplify the explanation used in this section. But the .NET types are improving over the time, so my recommendation is to access the official Microsoft documentation via https://docs.microsoft.com/en-us/dotnet/api/system.char?view=netframework-4.7.2 to consult the most recent declaration and details about the `System.Char` value type.

Listing 2-28. Declaration of the System.Char Value Type

```
/* C++/CLI (Common Language Infrastructure). */

[SerializableAttribute]
[ComVisibleAttribute(true)]
public value class Char : IComparable, IConvertible,
    IComparable<wchar_t>, IEquatable<wchar_t>
```

```
/* C# programming language. */

[System.Runtime.InteropServices.ComVisible(true)]
public struct Char : IComparable, IComparable<char>, IConvertible,
IEquatable<char>
```

The execution environment uses an instance of the `System.Char` value type to store the *code point* of the Unicode character. So, if we have a sequence of instances of `System.Char` value types, we would be much more productive working with them using the perspective of a set. That is the fundamental purpose of the string data type.

One of the lessons when we start learning about the Unicode standard is this one:

More than one code point can be needed to represent a single Unicode character.

Code Point, Code Unit and System.Char

In the Unicode standard, a set of bits with structured meaning of an integer is a *code unit*. For example, 8-bit (1 byte), 16-bit (2 bytes), and 32-bit (4 bytes) are all *code units*, and based on the size of each code unit, the standard defines the adequate *encoding form*. The symbol that represents the character, like the letter *A* or any other, is the *abstract character*, because we can have multiple code units that are associated with just one abstract character. When an abstract character is mapped with two or more code units, it is an *encoded Unicode character*. When expressed as a sequence of code units, the encoding form specifies how each integer in that sequence is organized to express a Unicode character, and one sequence can have multiple code units. This encoding form is named Unicode Transformation Format (UTF). The Unicode standard provides three encoding formats: UTF-8, UTF-16, and UTF-32 to 8-bit, 16-bit, and 32-bit, respectively, and the `System.Char` store the value as a UTF-16 code unit. The important point here is that all three encoding formats can provide access to a full range of encoded Unicode characters. They are fully interoperable, which is important to applications that need to work with different encoding formats. Also, the encoding forms can be transformed in any of the other two without loss of data and meaning. Here is an example of code units and code points. in three formats for the letter *A* (uppercase):

- Letter *A* (uppercase) using UTF-32: 0x00000041
- Letter *A* (uppercase) using UTF-16: 0x0041
- Letter *A* (uppercase) using UTF-8: 0x41

The Unicode standard defines each Unicode character with *one unique scalar number with 21 bits using UTF-32*, and this unique scalar number with 21-bits is the *code point*. But the encoding of each Unicode character is made with 16-bit (UTF-16). Because of this, a sequence of one or more 16-bit values (code units) can be used to represent one Unicode character. Each of these 16-bit values is a code unit, and the supported range for these 16 bits is from 0x0000 until 0xFFFF (65,535).

This can lead to peculiar aspects that we must be aware of. For example, when using an instance of `System.String`, the `String.Length` property returns the number of instances of the `System.Char` value type, not the number of Unicode characters. For our purposes, we need to remember the following:

- The .NET Framework adopted the Unicode standard as one of the pillars of the platform.
- `System.Char` is the fundamental built-in data type that is used to store Unicode character code units (16-bit).
- `System.Char` is also used for useful aggregated operations, such as getting the uppercase Unicode character representation.
- One instance of the `System.Char` value type does not necessarily mean one Unicode character. The one-by-one relationship is not a requirement; it just happens sometimes based on the rules of the Unicode encoding format.
- We must use the `System.String` reference type to process sequences of instances of the `System.Char` value type instead of dealing individually with them.
- The .NET Framework platform supports all three encoding formats: UTF-8, UTF-16, and UTF-32.

System.String

One of the main aspects of the `System.String` type is that the content of the string is treated in a special manner. One of the characteristics available is the *intern pool*, which is a table to each unique literal string that is declared or created programmatically by the code. Every time an instance of the CLR is created, the intern pool is created by that instance of the CLR. It is an optimization feature that the CLR uses to conserve string

storage. Every time we define a literal string value, the execution environment checks to see whether it is in the intern pool. If the literal string value is not in the intern pool, it is automatically inserted in it by the execution environment. If the managed process uses the same literal string value in any other part, the execution environment, instead of duplicating the literal string value, makes the instance of the string pointing to the literal string value that is in the intern pool. Even when used just one time, the literal string value is automatically inserted into the intern pool. If we want to check whether some string value is in the intern pool, we can use the `String.IsInterned()` static method. The method returns a reference to the interned string value or null if the string value it is not in the intern pool. The Listing 2-29 shows an example of the `String.IsInterned()` method:

Listing 2-29. Checking Whether the Literal String Value Is in the Intern Pool

```
/* C++/CLI (Common Language Infrastructure). */
String^ buffer{ "Automatically interned value." };
Console::WriteLine( "Is interned: {0}", ( String::IsInterned( buffer )
!= nullptr ).ToString() );

/* C# programming language. */
String buffer = "Automatically interned value.";
Console.WriteLine( "Is interned: {0}", ( String.IsInterned( buffer )
!=null ).ToString() );;
```

If the application reads the string values from a stream such as a file or network packet or from any device, we can use the `String.Intern()` static method to force the insertion of the string values in the intern pool, if we think that this strategy is adequate support to the performance of the operation. Listing 2-30 shows an "extreme" example, when all the content of the file is inserted into the intern pool. *Do not use this type of implementation in a real application.*

Remember that an intern pool is a kind of "special shared cache to literal string values," not to instances of `System.String`, and it is automatically managed by the instances of the execution environment, that is, instances of the CLR. For example, the `String.Empty` static field has the `""` empty string value, and this value is automatically created when an instance of the execution environment is created. The purpose of the

intern pool is to optimize the memory consumption. As we can remember, the .NET Framework adopted the Unicode standard, and each Unicode character (code point) can be represented by one or more 16-bit code units. Therefore, a simple multiplication operation using the number of characters in a sequence can alert us about how this strategy of optimization of memory consumption is important. Still, the use of the intern pool has side effects. While the instance of the execution environment is alive, the string values inserted in the intern pool are not released from memory, and the instance is available to the other managed processes in different application domains, even when our managed process that created these string values is not running anymore. This behavior is understandable when we remember that the managed execution environment uses a garbage collector mechanism. This means the allocated blocks of memory for the managed instances are not necessarily released when that associated variable is not in scope anymore; the reference will be released at a different moment by the garbage collector mechanism. If we identify that the intern pool is not adequate for the designed types in our project, we can have one or more specific assemblies not intern string values.

Listing 2-30. Inserting String Values in the Intern Pool

```
/* C++/CLI (Common Language Infrastructure) projection. */

StreamReader^ reader{ File::OpenText( "YourFile.txt" ) };
String^ buffer{};
while ( ( buffer = reader->ReadLine() ) != nullptr ) {

        buffer = String::Intern( buffer );
Console::WriteLine( "Line of text: {0}\n\nIs interned?: {1}", buffer,
(String::IsInterned( buffer ) != nullptr ).ToString() );

};

reader->Close(); /* Automatically calls IDisposable->Dispose() instance
method. */
reader = nullptr;
```

```
/* C# programming language. */

using ( StreamReader reader = File.OpenText( "YourFile.txt" ) ) {

        String buffer = null;
        while ( ( buffer = reader.ReadLine() ) != null ) {

                 buffer = String.Intern( buffer );
        Console.WriteLine( "Line of text: {0}\n\nIs interned?: {1}",
        buffer, ( String.IsInterned( buffer ) != null ).ToString() );

        };

        reader.Close();
};
```

The ToString() Method

The root type in the BCL is `System.Object`, and if we look at the members of the type, we can see the `ToString()` instance method. If we are working with any specialized environment such as .NET Azure applications, Console applications, Windows Forms applications, ASP.NET applications, ASP.Net Core applications, WPF applications, .NET Core applications, Windows Services implemented with .NET, applications created with .NET for UWP, applications created with Xamarin .NET technologies, and so on, we can see that all the objects need a `ToString()` instance method to create a representation of the state of the instance of that reference type or value type.

CHAPTER 3

Equality and Cloning Operations from VES Perspective

In this chapter, we will learn about two fundamental operations, `equals` and `clone`. As we learned in Chapter 1 and Chapter 2, the fundamental types `array` and `string` are part of any kind of application targeting an implementation of .NET. But some of the behaviors available within these types come from the VES and are expressed through methods and functionalities that are implicitly available. For example, references types and value types derive from the root reference type `System.Object`, implicitly or explicitly. So, reference types and value types inherit some of the fundamental behaviors designed and described by the CTS, which are implemented and enforced by the VES.

The Equality Operation

In this section, we talk about the equality operation from the perspective of the VES. We will show examples in C++/CLI, CIL, and the C# programming language.

As shown in Listing 3-1 in C++/CLI, we have two instances of the reference type `System.Object`, named `objA` and `objB`, and two references named `objC` and `objD`. The references `objC` and `objD` point to the instance pointed to by `objA`. The references `objA` and `objB` are independent instances.

Listing 3-1. Comparing Instances of Reference Types Using the Equality (==) Operator (Example in C++/CLI)

```
Object^ objA{ gcnew Object() };
Object^ objB{ gcnew Object() };

Console::WriteLine( "objA == objB: {0}", ( objA == objB ).ToString()  );
```


R. Villela, *Pro .NET Framework with the Base Class Library*, https://doi.org/10.1007/978-1-4842-4191-2_3

When we use the equality operator (==) for an instance of the type `System.Object`, C++/CLI emits an optimized instruction of the CIL set: `bne.un.s` (branch if unequal or unordered, short form). Listing 3-2 shows the CIL instructions for these operations.

Listing 3-2. CIL Instructions Used for Equality Operation and Branching Operation

```
.method assembly static int32 modopt([mscorlib]System.Runtime.
CompilerServices.CallConvCdecl)
        wmain() cil managed
{
  .vtentry 1 : 1
  // Code size         52 (0x34)
  .maxstack  2
  .locals ([0] object objB,
           [1] object objA,
           [2] string V_2,
           [3] bool V_3)

  IL_0000:  ldnull
  IL_0001:  stloc.1
  IL_0002:  ldnull
  IL_0003:  stloc.0
  IL_0004:  newobj     instance void [mscorlib]System.Object::.ctor()
  IL_0009:  stloc.1
  IL_000a:  newobj     instance void [mscorlib]System.Object::.ctor()
  IL_000f:  stloc.0
  IL_0010:  ldloc.1
  IL_0011:  ldloc.0
  IL_0012:  bne.un.s   IL_0017
  IL_0014:  ldc.i4.1
  IL_0015:  br.s       IL_0018
  IL_0017:  ldc.i4.0
  IL_0018:  stloc.3
  IL_0019:  ldloca.s   V_3
  IL_001b:  call       instance string [mscorlib]System.Boolean::ToString()
  IL_0020:  stloc.2
```

```
  IL_0021:  ldstr      "objA == objB: {0}"
  IL_0026:  ldloc.2
  IL_0027:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                object)
  IL_002c:  ldc.i4.1
  IL_002d:  call       void modopt([mscorlib]System.Runtime.CompilerServices.
                       CallConvCdecl) '?A0xbf94fd11.Pause'(bool)
  IL_0032:  ldc.i4.0
  IL_0033:  ret
} // end of method 'Global Functions'::wmain
```

We can get the same result of the example shown in Listing 3-2 using the `Object.ReferenceEquals()` static method. The purpose of this method is to compare two references and return a boolean value if these references point to the same instance. Listing 3-3 shows how to do this.

Listing 3-3. Comparing Instances of Reference Types Using the Object.ReferenceEquals() Static Method (Example in C++/CLI)

```
Object^ objA{ gcnew Object() };
Object^ objB{ gcnew Object() };

Console::WriteLine( "objA == objB: {0}", ( objA == objB ).ToString()  );

Console::WriteLine( "objA == objB: {0}", Object::ReferenceEquals( objA,
objB ).ToString()  );
```

Now, the C++/CLI projection obeys our choice and, instead of using an optimized instruction of the CIL set, uses the explicit call for the `Object.ReferenceEquals()` static method, as shown in Listing 3-4.

Listing 3-4. CIL Instructions Used with the Object.ReferenceEquals() Static Method

```
.method assembly static int32 modopt([mscorlib]System.Runtime.
CompilerServices.CallConvCdecl)
        wmain() cil managed
{
  .vtentry 1 : 1
  // Code size       69 (0x45)
```

```
  .maxstack  2
  .locals ([0] object objB,
           [1] object objA,
           [2] bool V_2,
           [3] bool V_3)
  IL_0000:  ldnull
  IL_0001:  stloc.1
  IL_0002:  ldnull
  IL_0003:  stloc.0
  IL_0004:  newobj     instance void [mscorlib]System.Object::.ctor()
  IL_0009:  stloc.1
  IL_000a:  newobj     instance void [mscorlib]System.Object::.ctor()
  IL_000f:  stloc.0
  IL_0010:  ldloc.1
  IL_0011:  ldloc.0
  IL_0012:  bne.un.s   IL_0017
  IL_0014:  ldc.i4.1
  IL_0015:  br.s       IL_0018
  IL_0017:  ldc.i4.0
  IL_0018:  stloc.3
  IL_0019:  ldstr      "objA == objB: {0}"
  IL_001e:  ldloca.s   V_3
  IL_0020:  call       instance string [mscorlib]System.Boolean::ToString()
  IL_0025:  call       void [mscorlib]System.Console::WriteLine(string, object)
  IL_002a:  ldloc.1
  IL_002b:  ldloc.0
  IL_002c:  call       bool [mscorlib]System.Object::ReferenceEquals(object,
                                                                     object)
  IL_0031:  stloc.2
  IL_0032:  ldstr      "objA == objB: {0}"
  IL_0037:  ldloca.s   V_2
  IL_0039:  call       instance string [mscorlib]System.Boolean::ToString()
  IL_003e:  call       void [mscorlib]System.Console::WriteLine(string, object)
  IL_0043:  ldc.i4.0
  IL_0044:  ret
} // end of method 'Global Functions'::wmain
```

We have another choice to perform the comparison of equality or inequality between the references: the `Equals()` instance method. Again, C++/CLI obeys our choice and emits the CIL instruction to call the `Equals()` instance method. Listing 3-5 shows how to do this.

Listing 3-5. Comparison of Instances of Reference Types Using the Equals() Instance Method

```
Object^ objA{ gcnew Object() };
Object^ objB{ gcnew Object() };

Console::WriteLine( "objA == objB: {0}", objA->Equals( objB ).ToString()  );
```

Listing 3-6 shows the excerpt of CIL code generated from the C++/CLI code in Listing 3-5.

Listing 3-6. CIL Using the Equals() Instance Method for Comparison of Instances of Reference Types

```
  .locals ([0] object objB,
           [1] object objA,
           [2] bool V_2)
  IL_0000:  ldnull
  IL_0001:  stloc.1
  IL_0002:  ldnull
  IL_0003:  stloc.0
  IL_0004:  newobj     instance void [mscorlib]System.Object::.ctor()
  IL_0009:  stloc.1
  IL_000a:  newobj     instance void [mscorlib]System.Object::.ctor()
  IL_000f:  stloc.0
  IL_0010:  ldloc.1
  IL_0011:  ldloc.0
  IL_0012:  callvirt   instance bool [mscorlib]System.Object::Equals(object)
  IL_0017:  stloc.2
  IL_0018:  ldstr      "objA == objB: {0}"
  IL_001d:  ldloca.s   V_2
  IL_001f:  call       instance string [mscorlib]System.Boolean::ToString()
  IL_0024:  call       void [mscorlib]System.Console::WriteLine(string, object)
```

But what happens with the same sequence of code when implemented with the C# programming language? We may get a pleasant surprise here. The implementation in C# is shown in Listing 3-7.

Listing 3-7. Comparison of Instances of Reference Types (Example in C#)

```
Object  objA = new Object();
Object  objB = new Object();

Console.WriteLine( "objA == objB: {0}", ( objA == objB ).ToString() );
```

The C# compiler emits another optimized instruction too but in this case uses the `ceq` (compare equal) instruction of the CIL set, as shown in the CIL code in Listing 3-8.

Listing 3-8. CIL Used for Comparison of Instances of Reference Types

```
  .locals init ([0] object objA,
                [1] object objB,
                [2] bool V_2)
  IL_0000:  nop
  IL_0001:  newobj     instance void [mscorlib]System.Object::.ctor()
  IL_0006:  stloc.0
  IL_0007:  newobj     instance void [mscorlib]System.Object::.ctor()
  IL_000c:  stloc.1
  IL_000d:  ldstr      "objA == objB: {0}"
  IL_0012:  ldloc.0
  IL_0013:  ldloc.1
  IL_0014:  ceq
  IL_0016:  stloc.2
  IL_0017:  ldloca.s   V_2
  IL_0019:  call       instance string [mscorlib]System.Boolean::ToString()
  IL_001e:  call       void [mscorlib]System.Console::WriteLine(string, object)
```

What about the other two scenarios? When we use the `Object.ReferenceEquals()` static method, the C# compiler ignores our choice and emits the same sequence of the CIL, using the `ceq` instruction. But, when we use the `Equals()` instance method, the C# compiler obeys our choice and emits the CIL sequence for calling the specific instance method. This is shown with the code in Listing 3-9.

Listing 3-9. Comparison of Instances of Reference Types Using the Object.ReferenceEquals() Static Method

```
Object objA = new Object();
Object objB = new Object();

Console.WriteLine( "objA == objB: {0}", Object.ReferenceEquals( objA, objB ).
ToString() );
```

In Listing 3-10 we can see the CIL code for the C# code shown in Listing 3-9.

Listing 3-10. CIL Code for Comparison of Instances of Reference Types Using the Object.ReferenceEquals() Static Method

```
.locals init ([0] object objA,
              [1] object objB,
              [2] bool V_2)
IL_0000:  nop
IL_0001:  newobj     instance void [mscorlib]System.Object::.ctor()
IL_0006:  stloc.0
IL_0007:  newobj     instance void [mscorlib]System.Object::.ctor()
IL_000c:  stloc.1
IL_000d:  ldstr      "objA == objB: {0}"
IL_0012:  ldloc.0
IL_0013:  ldloc.1
IL_0014:  ceq
IL_0016:  stloc.2
IL_0017:  ldloca.s   V_2
IL_0019:  call       instance string [mscorlib]System.Boolean::ToString()
IL_001e:  call       void [mscorlib]System.Console::WriteLine(string, object)
```

Listing 3-11 shows the C# code to use the `Equals()` instance method.

Listing 3-11. Comparison of Instances of Reference Types Using the Equals() Instance Method

```
Object objA = new Object();
Object objB = new Object();

Console.WriteLine( "objA == objB: {0}", objA.Equals( objB ).ToString() );
```

Listing 3-12 shows the CIL code for the C# code shown in Listing 3-11:

Listing 3-12. CIL Code for Comparison of Instances of Reference Types Using the Equals() Instance Method

```
  .locals init ([0] object objA,
                [1] object objB,
                [2] bool V_2)
IL_0000:  nop
IL_0001:  newobj     instance void [mscorlib]System.Object::.ctor()
IL_0006:  stloc.0
IL_0007:  newobj     instance void [mscorlib]System.Object::.ctor()
IL_000c:  stloc.1
IL_000d:  ldstr      "objA == objB: {0}"
IL_0012:  ldloc.0
IL_0013:  ldloc.1
IL_0014:  callvirt   instance bool [mscorlib]System.Object::Equals(object)
IL_0019:  stloc.2
IL_001a:  ldloca.s   V_2
IL_001c:  call       instance string [mscorlib]System.Boolean::ToString()
IL_0021:  call       void [mscorlib]System.Console::WriteLine(string, object)
```

The C# Compiler and the Object.ReferenceEquals() method

In this case, the implementation of the `Object.ReferenceEquals()` static method does this: `(objA == objB)`. This results in a call to the `Equals` type instance method. To the `Object.Equals()` instance method, most of the implementation is part VES (C++ code) and part BCL (C# code). These are excerpts of code from source files of the BCL.

Listing 3-13 shows excerpts of the implementation of the `System.Object ReferenceEquals()` static method, which is the `Equals()` instance method. Microsoft uses C# in the implementation. We can access these source code files at https://referencesource.microsoft.com/.

Listing 3-13. Excerpts of the System.Object Type Implementation

```
/* Object.cs reference equals method. */

[System.Runtime.Versioning.NonVersionable]
public static bool ReferenceEquals (Object objA, Object objB) {
        return objA == objB;
    }

/* Object.cs equals method. */

public virtual bool Equals(Object obj)
{
      return RuntimeHelpers.Equals(this, obj);
}

public static bool Equals(Object objA, Object objB)
{
        if (objA==objB) {
            return true;
        }
        if (objA==null || objB==null) {
            return false;
        }
        return objA.Equals(objB);
}
```

`RuntimeHelpers` is a static class with static methods that provide support to compilers. The `RuntimeHelpers.Equals()` static method is part of the type and is declared with the `extern` C# keyword to indicate that the method is implemented in some external component implemented in native (C++) code (like a DLL). The method implementation attribute with the enumeration `MethodImplOptions` type as an argument indicates how the method is implemented. In this case, with the value `MethodImplOptions.InternalCall`, the method is implemented within the CLR, which is part of the implementation of the VES. Listing 3-14 shows the declaration of the `RuntimeHelpers.Equals()` static method.

Listing 3-14. Declaration of the Equals() Instance Method

```
[MethodImplAttribute(MethodImplOptions.InternalCall)]
public new static extern bool Equals(Object o1, Object o2);
```

From the perspective of the C# compiler, an "extra" call to the `Object.ReferenceEquals()` static method is not a good option in the particular case of a custom type implementation that does not override the `Equals()` instance method. This is because, in the end, the called method is the foundational internal implementation in the VES. When compared with this "extra" call to the `Object.ReferenceEquals()` static method, the CIL instructions `ceq` (compare equal), `bne.un` (branch to target if unequal or unordered), and `bne.un.s` (branch to target if unequal or unordered, short form) are optimizations of the implementation of the VES. When we think about the case of the C++/CLI, the compiler considers that we know what we are doing and trusts, on most cases similar to this, our decisions.

This is why no compiler considers replaces an explicit call to the nonoverrided `Equals()` instance method standard implementation and can choose to adopt some basic strategic CIL instructions to reduce the path for the call of standard equality behavior. This is one of the most basic lessons when working with OOP (but not only) and inheritance, called the *shortest adequate path*. This can be viewed as a moderated decision considering the size of the assembly module (EXE, DLL, and so on) and runtime performance (processor cycles of execution and complexity of the assembly instructions). This reduces the overload of the just-in-time (JIT) and ahead-of-time (AOT) mechanisms, the overload about the garbage collector (GC) mechanism, memory consumption, and other intrinsic aspects.

Equality and Inequality with Value Types

In this section, we present examples of the equality on value types and start with the C# programming language. By definition, value types (as we can guess) are compared by value, which means that to be considered equal, two or more instances of value types must have the same value.

In Listing 3-15, we have two instances of the *value type* of `System.Int32` named `vltA` and `vltB`. The instance `vltB` is created using a copy of the value in the instance `vltA`, which means that the value of the instance `vltA` is copied and used as a start value for the instance `vltB`, but the two are distinct instances. When we compare `vltA` with `vltB`, the result is true, because both have the same value.

Listing 3-15. Comparison of Instances of Value Types for Equality

```
Int32 vltA = 100;
Int32 vltB = vltA;

Console.WriteLine( "vltA == vltB: {0}", ( vltA == vltB ).ToString() );
```

The sequence emitted by the C# compiler is similar to what we saw before. Again, the compiler uses the `ceq` CIL instruction (and now we understand why). This is shown in the CIL code in Listing 3-16.

Listing 3-16. CIL for Equality Operation: Using the ceq (Compare Equal) Intermediate Instruction

```
.locals init ([0] int32 vltA,
              [1] int32 vltB,
              [2] bool V_2)
IL_0000:  nop
IL_0001:  ldc.i4.s   100
IL_0003:  stloc.0
IL_0004:  ldloc.0
IL_0005:  stloc.1
IL_0006:  ldstr      "vltA == vltB: {0}"
IL_000b:  ldloc.0
IL_000c:  ldloc.1
IL_000d:  ceq
IL_000f:  stloc.2
IL_0010:  ldloca.s   V_2
IL_0012:  call       instance string [mscorlib]System.Boolean::ToString()
IL_0017:  call       void [mscorlib]System.Console::WriteLine(string, object)
```

Using the ReferenceEquals() Static Method

When using the `ReferenceEquals()` static method, the result will not be the same.

We can understand this with the C# code example in Listing 3-17 and in the CIL example in Listing 3-18.

Listing 3-17. Comparison of Instances of Value Types Using the Object. ReferenceEquals() Static Method

```
Int32 vltA = 100;
Int32 vltB = vltA;

Console.WriteLine( "vltA == vltB: {0}", Object.ReferenceEquals( vltA, vltB ).
ToString() );
```

As shown in the CIL code in Listing 3-18, the sequence emitted uses a box instruction that we discussed in Chapter 2. But the rest of the sequence is the same as the CIL code in Listing 3-16.

Listing 3-18. CIL to Instances of Value Types When Boxed

```
.locals init ([0] int32 vltA,
              [1] int32 vltB,
              [2] bool V_2)
IL_0000:  nop
IL_0001:  ldc.i4.s   100
IL_0003:  stloc.0
IL_0004:  ldloc.0
IL_0005:  stloc.1
IL_0006:  ldstr      "vltA == vltB: {0}"
IL_000b:  ldloc.0
IL_000c:  box        [mscorlib]System.Int32
IL_0011:  ldloc.1
IL_0012:  box        [mscorlib]System.Int32
IL_0017:  ceq
IL_0019:  stloc.2
IL_001a:  ldloca.s   V_2
```

```
IL_001c:  call       instance string [mscorlib]System.Boolean::ToString()
IL_0021:  call       void [mscorlib]System.Console::WriteLine(string, object)
```

Verifying That the Comparison Result Is False

Even with the values being the same, the comparison result is false! But, how is that?

Here we have an especially important sequence and an issue. Remember that with the instances of boxed value types we have two instances of the reference type `System.Object`. The `ceq` does not check whether the instances are boxed value types or not, and the `ceq` instruction compares the two objects that are popping out of the top of the stack. In this case, they are the two references pointing to different instances of `System.Object`.

If we create this sequence using C++/CLI, the behavior and result are the same as explained, even if `bne.un.s` was used instead of `ceq`. In this case, the compiler obeys our choices and will trust our judgment, emitting the call to the specific method. Listing 3-19 shows the code in C++/CLI, and Listing 3-20 shows the CIL code for the same scenario being discussed.

Listing 3-19. Comparison of Boxed Instances of Value Types Using the ReferenceEquals() Static Method

```
Int32 vltA{ 100 };
Int32 vltB{ vltA };

Console::WriteLine( "vltA == vltB: {0}", Object::ReferenceEquals
( vltA, vltB ).ToString() );
```

Listing 3-20. CIL to Instances of Value Types When Boxed

```
.locals ([0] int32 vltA,
         [1] int32 V_1,
         [2] int32 vltB,
         [3] bool V_3)
IL_0000:  ldc.i4.0
IL_0001:  stloc.1
IL_0002:  ldc.i4.s   100
IL_0004:  stloc.0
IL_0005:  ldloc.0
```

```
IL_0006:  stloc.2
IL_0007:  ldloc.0
IL_0008:  box        [mscorlib]System.Int32
IL_000d:  ldloc.2
IL_000e:  box        [mscorlib]System.Int32
IL_0013:  call       bool [mscorlib]System.Object::ReferenceEquals(object,
                                                                   object)
IL_0018:  stloc.3
IL_0019:  ldstr      "vltA == vltB: {0}"
IL_001e:  ldloca.s   V_3
IL_0020:  call       instance string [mscorlib]System.Boolean::ToString()
IL_0025:  call       void [mscorlib]System.Console::WriteLine(string, object)
```

Creating a Custom Type Without Overriding the Equals Method and the Equality and Inequality Operators

When designing a custom type, we must understand the context of the type and define which rules make two instances equal.

For the hypothetical `Person` custom type, here we have a basic experimental implementation as a starting point. Listing 3-21 shows the implementation of the `Person` custom type using C++/CLI.

Listing 3-21. Person Custom Type, C++/CLI Implementation

```
public ref class Person {
      protected:
            StringBuilder ^ _name{};

      public:
            Person(): _name( gcnew StringBuilder() ) {
                  return;
      };

      Person( String^ newName ): Person() {
            this->Name = newName;
            return;
      };
```

```
    public:
        property String^ Name {
            void set( String^ newName ) {
                this->_Validate_Name( newName );
                return;
            };
            String^ get() { return this->_name->ToString(); };
        };
    private:
        void _Validate_Name( String^ _proposedName ) {
            if ( _proposedName != nullptr ) this->_name->Clear()->
            Append( _proposedName );
            else this->_name->Clear()->Append( String::Empty) ;
            return;
        };
};
```

Listing 3-22 shows the implementation of the Person custom type using C#.

Listing 3-22. Person Custom Type, C# Implementation

```
public class Person {

    protected StringBuilder _name;

    public Person(){
        _name = new StringBuilder();
    }

    public Person(string newName) : this() {
        Name = newName;
    }
```

```
    public string Name {
        set { _Validate_Name(value); }

        get { return _name.ToString(); }
    }

    private void _Validate_Name(string proposedName) {

        if (proposedName != null) _name.Clear().Append(proposedName);
        else _name.Clear().Append(string.Empty);
    }
}
```

Even though we did not explicitly declare the `Person` reference type inherited from the `System.Object` root reference type, this implicitly happens. Listing 3-23 shows this excerpt of the emitted CIL by both compilers, C# and C++/CLI.

Listing 3-23. CIL Declaration of the Person Custom Type

```
.class public auto ansi beforefieldinit Person
       extends [mscorlib]System.Object
{
} // end of class Person
```

When implementing using C++/CLI, if we write the declaration `ref class MyRefType: System::Object`, the compiler emits a warning about ambiguous `System::Object` instances in many points of the code, for example, methods returning the `System::Object`. To resolve this issue, remove `System::Object` from the declaration of the custom type: `ref class MyRefType`. This issue does not happen with value types because value types cannot explicitly inherit from other managed types, except from interfaces. Examples of these errors are shown in Figure 3-1.

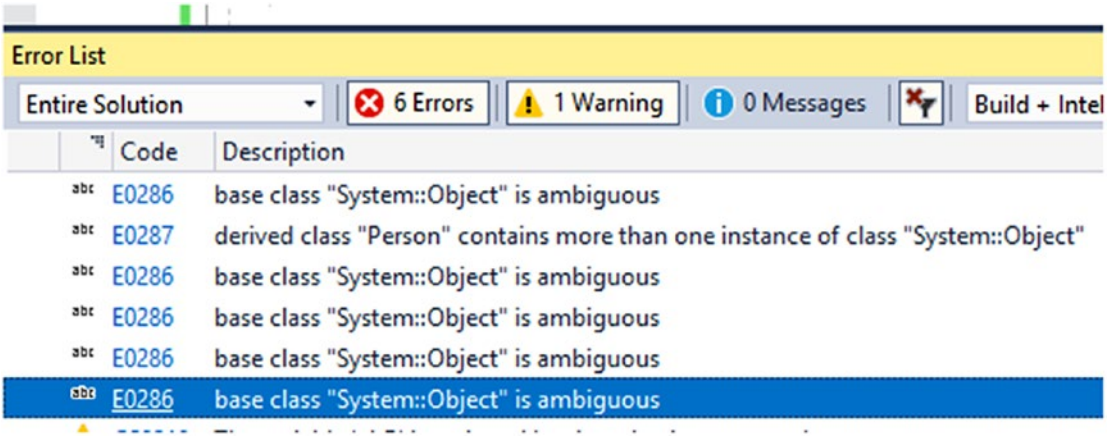


Figure 3-1. *Compiler error when using System::Object in a declaration of a type in C++/CLI*

As we learned from Chapter 2, if we try to use instance methods like `ToString()`, `Equals()`, `GetHashCode()`, `GetType()`, and `MemberwiseClone()`, we will automatically obtain not only these standard instance methods but also the standard behaviors from these instance methods without having to write a single line of code for our custom reference type or value type, for example. Even when using fundamental operators such as equality (==) and inequality (!=), they are already available. For example, if we use the equality (==) operator, the `Equals()` instance method, and the `Object.ReferenceEquals()` static method with the `Person` custom reference type, we will get the same results from the C++/CLI and C# programming languages and the CIL emitted in regard to the `bne.un`, `bne.un.s`, and `ceq` instructions. This is one of the most powerful aspects of the standardization of the platform. Within this basic example we are seeing language-independent, hardware-independent, and operating system–agnostic intermediate code. In regard to the intermediate code instructions, the compilers have the freedom to choose different options considering the developer profile, just to cite a few of the elements in the CLI specification.

Listing 3-24 shows an example of using the `Person` custom reference type using C++/CLI and the Listing 3-26 shows the same example but using C# programming language. The Listing 3-25 shows the CIL generated from this C++/CLI implementation, and the Listing 3-27 shows the CIL generated from this C# implementation.

Listing 3-24. C++/CLI Use of Person Custom Reference Type

```
Person^ prsA{ gcnew Person( "A" ) };
Person^ prsB{ gcnew Person( "B" ) };

Console::WriteLine( prsA == prsB );
```

Listing 3-25. Excerpt of CIL Code from the C++/CLI in Listing 3-24

```
.locals ([0] uint8 V_0,
         [1] class Person prsB,
         [2] class Person prsA,
         [3] int32 V_3)
IL_0000:  ldc.i4.0
IL_0001:  stloc.3
IL_0002:  ldnull
IL_0003:  stloc.2
IL_0004:  ldnull
IL_0005:  stloc.1
IL_0006:  ldstr      "A"
IL_000b:  newobj     instance void Person::.ctor(string)
IL_0010:  stloc.2
IL_0011:  ldstr      "B"
IL_0016:  newobj     instance void Person::.ctor(string)
IL_001b:  stloc.1
IL_001c:  ldloc.2
IL_001d:  ldloc.1
IL_001e:  bne.un.s   IL_0024
IL_0020:  ldc.i4.1
IL_0021:  stloc.0
IL_0022:  br.s       IL_0026
IL_0024:  ldc.i4.0
IL_0025:  stloc.0
IL_0026:  ldloc.0
IL_0027:  call       void [mscorlib]System.Console::WriteLine(bool)
```

Listing 3-26. C# Use of the Person Custom Reference Type

```
Person prsA = new Person( "A" );
Person prsB = new Person( "B" );

Console.WriteLine( prsA == prsC );
```

Listing 3-27. Excerpt of CIL Code from the C# in Listing 3-26

```
.locals init ([0] class Person prsA,
         [1] class Person prsB)
IL_0000:  nop
IL_0001:  ldstr      "A"
IL_0006:  newobj     instance void Person::.ctor(string)
IL_000b:  stloc.0
IL_000c:  ldstr      "B"
IL_0011:  newobj     instance void Person::.ctor(string)
IL_0016:  stloc.1
IL_0017:  ldloc.0
IL_0018:  ldloc.1
IL_0019:  ceq
IL_001b:  call       void [mscorlib]System.Console::WriteLine(bool)
```

Compilers Do Not Ignore Calls to the Equals() Method

Now think about this: why do the compilers (and this applies to compilers of other programming languages) not ignore the calls to the `Equals()` instance method, like the C# compiler did with the call for the `Object.ReferenceEquals()` static method?

Each type in the contextualized type model (or type hierarchy), being a reference type or a value type, must have just one meaning for the equality behavior. Internally the implementation of the equality (`==`) and inequality (`!=`) operators call the implementation of the `Equals()` instance method for that specific type. The `Equals()` method is designed as an instance method and to be customizable by any managed type implementation, if necessary. This is the standard architecture and engineering model designed and described by CTS, structurally represented on the metadata, implemented by the VES, and adopted from the ground up by the BCL and FCL. So, anyone who is interested in writing libraries and components must understand and accept this aspect of the standard model. For example, if our library has a root custom type `Person` (or any other root custom type), this root type must have some standard contextualized meaning for the equality behavior. Of course, derived types can include their own specialized equality rules, but in the end, the contextualized type model as a representative description and implementation will be a composition of focused contextualized rules,

not a set of arbitrary rules accumulated in a few types, methods, and properties, with poor ways to customize when necessary and not representing the meaning of the type within the context.

Listing 3-28 and Listing 3-29 show a basic implementation of the `Equals()` instance method. As we learned in Chapter 2, when we override the `Equals()` instance method, we must override the `GetHashCode()` instance method. If we do not have a specialized algorithm or library for the hash function, we should use the standard one provided by `Object.GetHashCode()`. The implementation of the equality (`==`) and inequality (`!=`) operators shows the use of the `Equals()` instance method. If our custom type is designed to support other operations such as less-than (`<`), greater-than (`>`), and so on, our custom type must implement the adequate operators.

Listing 3-28. Custom Type with the Equals() Instance Method Implemented in C++/CLI

```
public ref class Person {

      protected:
            StringBuilder ^ _name{};

      public:
            Person(): _name( gcnew StringBuilder() ) {
                  return;
            };

            Person( String^ newName ): Person() {
                  this->Name = newName;
                  return;
            };

      public:
            property String^ Name {
                  void set( String^ newName ) {
                        this->_Validate_Name( newName );
                        return;
                  };
```

```
            String^ get() { return this->_name->ToString(); };
        };

    public:
        virtual Boolean Equals( Object^ anotherPerson ) override {
            return this->__is_equal_Person( dynamic_cast<Person^>(
            anotherPerson ) );
        };
        virtual Int32 GetHashCode() override { return this-
        >GetHashCode(); };

        Boolean operator==( Person^ other ) {
            return this->Equals( other );
        };

        Boolean operator!=( Person^ other ) {
            return ( !this->Equals( other ) ) ;
        };

    private:
        void _Validate_Name( String^ _proposedName ) {
            if ( _proposedName != nullptr ) this->_name->Clear()-
            >Append( _proposedName );
        else this->_name->Clear()->Append( String::Empty );
        return;
    };

        Boolean __is_equal_Person( Person^ otherPerson ) {

            return  (( Person::ReferenceEquals( this, otherPerson ) ) &&
            ( this->Name ==  otherPerson->Name ) );
        };
};
```

Listing 3-29. Custom Type with the Equals() Instance Method Implemented in C#

```
public class Person {
      protected StringBuilder _name;
      public Person() {
            this._name = new StringBuilder();

            return;
      }
      public Person( string _newName ) : this() {
            this._name = _newName;
            return;
      }
      public string Name {
            set {
                  this._validate_Name( value );

                  return;
            }
            get { return this._name.ToString(); }
      }

public override bool Equals( object anotherPerson ) {
            return this.__is_equal( ( anotherPerson as Person ) );
      }

      public override int GetHashCode() { return this.GetHashCode(); }
      public static bool operator ==( Person me, Person other ) {
            return me.Equals( other );
      }

      public static bool operator !=( Person me, Person other ) {
            return !me.Equals( other ) ;
      }

private static string  _validate_Name( string _proposedName ) {
            return ( ( !Person._is_Name_Empty( _proposedName ) ? String.
            Intern( _proposedName ) : String.Empty ) );
      }
```

```
        private bool __is_equal( Person otherPerson ) {
                return ( ( Person.ReferenceEquals( this, otherPerson ) ) && (
                this.Name == otherPerson.Name ) );

        }
}
```

The ceq Instruction and bne.un and bne.un.s CIL Instructions

We can obtain the same result using any of these instructions, but there are distinct aspects in how the VES works with them. First, the CIL instruction set is organized into logical and formal categories by purpose, for example, the flow control instructions. However, there are other factors for this categorization, and two of these factors are the type and size of the parameter.

In the CIL instruction set, there are instructions that are classified as *long-parameter form* and *short-parameter form*. These particular instructions deal with parameters of the integer and unsigned integer types. The instructions that deal with parameters that require 4-byte integers are the long-parameter form of instructions. The instructions that deal with parameters that require 1-byte integers are the short-parameter form of instructions and have the `.s` suffix, used when the arguments to the parameters are in the range -128 to 127 for signed parameters and from 0 to 255 for unsigned parameters. `bne.un.s` is one example in that set of instructions.

`bne.un` and `ceq` are in the long-parameter form set. In fact, we can use the long-parameter form instructions for most scenarios in which the short-parameter form instructions are used, but this is a waste of resources, and some of the most direct consequences are the growth of the size of modules, load time, processing time, memory consumption, energy consumption, and, probably, poor programming practices, independent of the chosen programming language.

How the VES Deals with ceq, bne.un, and bne.un.s

`ceq` is in the *logical conditional check* instruction set. The instruction occupies 4 bytes in the metadata encoding and requires two objects on the stack to compare. The result is an `int32` value type, with 1 for true and 0 (zero) for false. There is no branching performed by the instruction, and we need to use other instructions like `brfalse.s` and `br.s`, for example, to do the branching if required, and how the arguments are taken

from the stack, no arguments to the instruction are included in the CIL stream. From the resulting value, an instance of `System.Boolean` can be created when required by code and with the appropriate true or false value.

In Listing 3-30 we have C# code where we can read about the effect of the boxed instances.

Listing 3-30. The Effect of Boxed Instances of Value Types and the ceq Intermediate Instruction

```
Int32 vltA = 100;
Int32 vltB = vltA;

Boolean isEqual = Object.ReferenceEquals( vltA, vltB );

if ( isEqual ) Console.WriteLine( "vltA == vltB: {0}", isEqual.ToString() );
else Console.WriteLine( "Bla, Bla, Bla...");
```

In Listing 3-31 we have some CIL code that is generated by the C# compiler.

Listing 3-31. Excerpt of CIL Code Generated from the Code in Listing 3-30

```
.locals init ([0] int32 vltA,
              [1] int32 vltB,
              [2] bool V_2)
IL_0000:  nop
IL_0001:  ldc.i4.s   100
IL_0003:  stloc.0
IL_0004:  ldloc.0
IL_0005:  stloc.1
IL_0006:  ldstr      "vltA == vltB: {0}"
IL_000b:  ldloc.0
IL_000c:  box        [mscorlib]System.Int32
IL_0011:  ldloc.1
IL_0012:  box        [mscorlib]System.Int32
IL_0017:  ceq
IL_0019:  stloc.2
IL_001a:  ldloca.s   V_2
IL_001c:  call       instance string [mscorlib]System.Boolean::ToString()
IL_0021:  call       void [mscorlib]System.Console::WriteLine(string, object)
```

In Listing 3-32 we have the same sequence in C++/CLI and the generated intermediate code. As we can see, the fundamental difference is, as explained, that C++/CLI is considering the code we write as a top priority and doesn't automatically replace our code with one specific standard sequence.

In Listing 3-32 we have some C++/CLI code where we can read about the effect of the boxed instances.

Listing 3-32. The Effect of Boxed Instances of Value Types and the Object.ReferenceEquals() Static Method

```
Int32 vltA{ 100 };
Int32 vltB{ vltA };

Boolean isEqual{ Object::ReferenceEquals( vltA, vltB ) };

if ( isEqual ) Console::WriteLine( "vltA == vltB: {0}", isEqual.ToString() );
else Console::WriteLine( "Bla, Bla, Bla..." );

Console::ReadLine();
```

In Listing 3-33 we have some CIL code that is generated by the C++/CLI compiler.

Listing 3-33. Excerpt of CIL Code Generated from the Code in Listing 3-32

```
.locals ([0] bool isEqual,
         [1] int32 vltA,
         [2] int32 V_2,
         [3] int32 vltB,
         [4] bool V_4)
IL_0000:  ldc.i4.0
IL_0001:  stloc.2
IL_0002:  ldc.i4.s   100
IL_0004:  stloc.1
IL_0005:  ldloc.1
IL_0006:  stloc.3
IL_0007:  ldloc.1
IL_0008:  box        [mscorlib]System.Int32
IL_000d:  ldloc.3
IL_000e:  box        [mscorlib]System.Int32
```

```
IL_0013:  call       bool [mscorlib]System.Object::ReferenceEquals(object,
                     object)
IL_0018:  stloc.0
IL_0019:  ldloc.0
IL_001a:  brfalse.sIL_0032
IL_001c:  ldloc.0
IL_001d:  stloc.s    V_4
IL_001f:  ldstr      "vltA == vltB: {0}"
IL_0024:  ldloca.s   V_4
IL_0026:  call       instance string [mscorlib]System.Boolean::ToString()
IL_002b:  call       void [mscorlib]System.Console::WriteLine(string, object)
IL_0030:  br.s       IL_003c
IL_0032:  ldstr      "Bla, Bla, Bla..."
IL_0037:  call       void [mscorlib]System.Console::WriteLine(string)
IL_003c:  call       string [mscorlib]System.Console::ReadLine()
```

Comparing the Behavior of the C# and C++/CLI Compilers Regarding the Equality Operator

For the equality operator (==), what is the behavior of the C# and C++/CLI compilers?

First, we see what happen when using the C++/CLI compiler. This is shown in Listing 3-34. The Listing 3-35 shows the CIL from this C++/CLI implementation.

Listing 3-34. The Equality (==) Operator with Instances of Value Types

```
Int32 vltA{ 100 };
Int32 vltB{ vltA };

Boolean isEqual{ vltA == vltB };

if ( isEqual ) Console::WriteLine( "vltA == vltB: {0}", isEqual.ToString() );
else Console::WriteLine( "Bla, Bla, Bla..." );

Console::ReadLine();
```

Listing 3-35. Excerpt of CIL Code Generated from the Code in Listing 3-34

```
.locals ([0] bool isEqual,
         [1] int32 vltA,
         [2] int32 V_2,
         [3] int32 vltB,
         [4] bool V_4)
IL_0000:  ldc.i4.0
IL_0001:  stloc.2
IL_0002:  ldc.i4.s   100
IL_0004:  stloc.1
IL_0005:  ldloc.1
IL_0006:  stloc.3
IL_0007:  ldloc.1
IL_0008:  ldloc.3
IL_0009:  bne.un.s   IL_000e
IL_000b:  ldc.i4.1
IL_000c:  br.s       IL_000f
IL_000e:  ldc.i4.0
IL_000f:  stloc.0
IL_0010:  ldloc.0
IL_0011:  brfalse.s  IL_0029
IL_0013:  ldloc.0
IL_0014:  stloc.s    V_4
IL_0016:  ldstr      "vltA == vltB: {0}"
IL_001b:  ldloca.s   V_4
IL_001d:  call       instance string [mscorlib]System.Boolean::ToString()
IL_0022:  call       void [mscorlib]System.Console::WriteLine(string, object)
IL_0027:  br.s       IL_0033
IL_0029:  ldstr      "Bla, Bla, Bla..."
IL_002e:  call       void [mscorlib]System.Console::WriteLine(string)
IL_0033:  call       string [mscorlib]System.Console::ReadLine()
```

In Listing 3-36 we have the example of the equality (==) operator with value types. The Listing 3-37 shows the CIL generated from this C# implementation.

Listing 3-36. The Equality (==) Operator with Instances of Value Types

```
Int32 vltA = 100;
Int32 vltB = vltA;

Boolean isEqual = ( vltA == vltB );

if ( isEqual ) Console.WriteLine( "vltA == vltB: {0}", isEqual.ToString() );
else Console.WriteLine( "Bla, Bla, Bla...");

Console.ReadLine();
```

Listing 3-37. Excerpt of CIL Code Generated from the Code in Listing 3-36

```
.locals init ([0] int32 vltA,
              [1] int32 vltB,
              [2] bool isEqual,
              [3] bool V_3)
IL_0000:  nop
IL_0001:  ldc.i4.s   100
IL_0003:  stloc.0
IL_0004:  ldloc.0
IL_0005:  stloc.1
IL_0006:  ldloc.0
IL_0007:  ldloc.1
IL_0008:  ceq
IL_000a:  stloc.2
IL_000b:  ldloc.2
IL_000c:  stloc.3
IL_000d:  ldloc.3
IL_000e:  brfalse.s  IL_0024
IL_0010:  ldstr      "vltA == vltB: {0}"
IL_0015:  ldloca.s   isEqual
IL_0017:  call       instance string [mscorlib]System.Boolean::ToString()
IL_001c:  call       void [mscorlib]System.Console::WriteLine(string, object)
IL_0021:  nop
```

```
IL_0022:  br.s       IL_002f
IL_0024:  ldstr      "Bla, Bla, Bla..."
IL_0029:  call       void [mscorlib]System.Console::WriteLine(string)
IL_002e:  nop
IL_002f:  call       string [mscorlib]System.Console::ReadLine()
```

With the generated intermediate code from C++/CLI, the first difference is the use of the `bne.un.s` intermediate instruction. `bne.un` and `bne.un.s`, the short form of instruction, are in the comparative branching instruction set and occupy only 1 byte in the metadata encoding and can work with `int8` (short-parameter form) and `int32` (long-parameter form). These instructions transfer the control when `value1` is not equal to `value2`. All integer values are interpreted as an unsigned integer; this means the MSB is not treated as the sign bit and just as a data bit in the sequence. If we are curious about these instructions, we can use the types in the namespace `System.Reflection.Emit`. For example, the reference type `System.Reflection.Emit.OpCodes` has static read-only fields for the CIL set. The excerpts in C# and C++/CLI show the size of some instructions, as described in the CLI specification and available in any implementation of .NET. If we want to play with the CIL for these instructions, Listing 3-38 shows the sample code in intermediate language. These files are in the folder for this chapter.

Listing 3-38. CIL of the File Example_Ch_03_00.il

```
.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch03 {
}

.method static public void MyEntryPointMethod() cil managed {

.locals init (
      int32 vltA,
      int32 vltB,
      bool isEqual,
      bool result
)
.entrypoint
```

```
/* Load constant. */
ldc.i4.s            0x64

/* Store. */
stloc.s            vltA

ldloc.s            vltA
stloc.s            vltB

/* The values for comparison. */
ldloc.s    vltA
ldloc.s    vltB

ceq

stloc.s    isEqual
ldloc.s    isEqual

/* Branch if false. */
brfalse.s  IL_00

ldstr      "vltA == vltB: {0}"
ldloca.s   isEqual
call       instance string [mscorlib]System.Boolean::ToString()
call       void [mscorlib]System.Console::WriteLine(string,  object)

br.s       IL_Final            /* Branch unconditionally. */

IL_00:

ldstr      "Bla, Bla, Bla..."
call       void [mscorlib]System.Console::WriteLine(string)

IL_Final:

call       string [mscorlib]System.Console::ReadLine()
pop

ret

}
```

The next section is about two fundamental operations that sometimes are confused in their purposes and implementation forms.

Clone vs. Copy

An interesting aspect is that rarely we see code for publicly cloning the state of an instance and using this cloned state as a starting point for the values of a new instance. So, why does the `Object.MemberwiseClone()` instance method exist, and is it inheritable? We must remember and understand that the CTS uses many aspects of object-oriented programming (OOP), component-based design, and design by contract programming as structural and behavioral elements to shape the contextualized type model (type hierarchy) of the CTS and VES, so certain elements exist for use in conjunction with others, and not as an isolated piece. The `Object.MemberwiseClone()` instance method is one of these nonisolated pieces.

The `Object.MemberwiseClone()` instance method default implementation does a *shallow copy* by creating a new object. This new object is filled with a copy of the value of all nonstatic fields of the source instance. The fundamental rules to the shallow copy stand are as follows:

- Perform a bit-by-bit copy for each field with a value type.
- The reference stored in the field is copied, not the instance pointed by the reference.

If we look at the purpose of the `Object.MemberwiseClone()` instance method, we will learn two things.

- The purpose of the `Object.MemberwiseClone()` instance method is to support the implementation of the `ICloneable` interface and not to use random and isolated cloning operations. Therefore, we do not see the method being used often.
- The standard method is implemented in the C++ programming language code.
- The standard method has protected member access; it is inheritable.

The ICloneable Interface

The purpose of this interface is to support more complex cloning operations, beyond what is provided by default by the `Object.MemberwiseClone()` instance method implementation or an inherited and customized implementation of it. The interface has only one method, and we can guess the name of it: the `Clone()` method.

Considering our custom `Person` reference type for a cloning operation, if we try to use the inherited `Object.MemberwiseClone()` instance method implementation from the outside of the implementation body of the class, we cannot do it, as shown in Listing 3-39 with C++/CLI and in Listing 3-40 with the C# programming language.

Listing 3-39. Trying to Access a Protected Member MemberwiseClone() Using C++/CLI

```
Person^ prsA{ gcnew Person( "A" ) };

/* ERROR: The protected method is not designed for public access and use. */

Person^ mirror = prsA->MemberwiseClone();
```

Listing 3-40. Trying to Access a Protected Member MemberwiseClone() Using C#

```
Person prsA = new Person( "A" );

/* ERROR: The protected method is not designed for public access and use. */

Person mirror = prsA.MemberwiseClone();
```

If the design of the `Person` custom reference type or any other custom managed type is in question, considering the inclusion of the clone operation is much more adequate, implement the `ICloneable` interface and a specialized `MemberwiseClone()` implementation to support the standard shallow copy. Along with this, keep the uniformization of what is expected from these two behaviors. But there is a recommendation about the implementation of the `ICloneable.Clone()` method that says to not make this implementation with public member access. Why? Because the implementation technique used for the clone operation can use the deep copy or the shallow copy and the implementors of the operation do not have to follow a predictable rule, we cannot be sure that the cloning operation was implemented based on a standard (predictable) set of rules. The issue with this is that it is much more difficult for a library or component engineer to create a more flexible set of algorithms because they cannot know whether the contextualized type model (*type hierarchy*) for a set of types, has opted to use the shallow copy or the deep copy with the cloning operation. For example, type `B` has opted to use the shallow copy, but the type `R` indirectly inherited from type `B`, has opted to use the deep copy, but both have implemented the `ICloneable` interface because this was established as a rule by the designer of the set of types.

The array type is declared as shown in Listing 3-41.

Listing 3-41. Declaration of the System.Array Abstract Reference Type

```
[System.Runtime.InteropServices.ComVisible(true)]
public abstract class Array : ICloneable, System.Collections.IList, System.
Collections.IStructuralComparable, System.Collections.IStructuralEquatable
```

`ICloneable` is one of the interfaces implemented by the type. Internally the `Array.Clone()` public instance method performs a call for the protected `Object.MemberwiseClone()`. In fact, if we read about the `Array.MemberwiseClone()` implementation, it is not customized; it is the inherited implementation of `Object.MemberwiseClone()` without any modification, and to obtain this standard behavior for a custom managed type, we only need to declare a custom type that is derived from `System.Object`. The examples of this standard behavior are shown in Listing 3-42 and Listing 3-43 with C++/CLI and in Listing 3-44 and Listing 3-45 with the C# programming language.

Listing 3-42. Declaration of Person Custom Type Using C++/CLI

```
public ref class Person {};
```

Listing 3-43. The Effect of the Use of Clone Operation with Arrays (Example Using C++/CLI)

```
::cli::array<Person^>^ people = { gcnew Person( "A" ), gcnew Person( "B" ),
gcnew Person( "C" ) };

::cli::array<Int32>^ numbers = { 0i32, 1i32, 2i32, 3i32 };

::cli::array<Person^>^ clonePeople =
safe_cast<::cli::array<Person^>^>( people->Clone() );

::cli::array<Int32>^ cloneNumbers =
safe_cast<::cli::array<Int32>^>( numbers->Clone() );
```

Listing 3-44. Declaration of Person Custom Type Using C#

```
public class Person {}
```

Listing 3-45. The Effect of the Use of Clone Operation with Arrays (Example Using C#)

```
Person[] people = { new Person( "A" ), new Person( "B" ), new Person( "C" ) };

Int32[] numbers = { 0, 1, 2, 3 };

Person[] clonePeople = ( ( Person[] ) people.Clone() );

Int32[] cloneNumbers = ( ( Int32[] ) numbers.Clone() );
```

ICloneable Interface Implementation on String Type

As showed in the Listing 3-46, the declaration of the System.String includes the ICloneable interface.

Listing 3-46. Declaration of System.String Including the ICloneable Interface

```
[System.Runtime.InteropServices.ComVisible(true)]
public sealed class String : ICloneable, IComparable, IComparable<string>,
IConvertible, IEquatable<string>, System.Collections.Generic.IEnumerable<char>
```

The implementation of the clone operation just returns a reference to itself: `return this`. If we need an independent copy of the content of an instance of `string`, we need to use `String.Copy` or `String.CopyTo`, for example. The proposed reason for that choice is that an instance of `string` is immutable. As we can remember, one of the things that I described about the implementation of the `ICloneable.Clone()` method is that there is a recommendation to not expose publicly an implementation of it because there is no uniformization about which is the standard implementation technique used for the cloning operation, shallow copy, or deep copy. By default, the types of the BCL and .NET FCL have opted for the shallow copy, but as we can see, this is not a rule, even for the most fundamental types.

Implementing the ICloneable Interface on Custom Types

The implementation of the `ICloneable` interface can be viewed as a signature that the type can be used in a cloning operation. For example, instead of considering the `Clone()` method, we can create an algorithm that checks whether the type has the `ICloneable` interface. If that is true, we can access the state of the instance and make a shallow copy or a deep copy using reflection, conforming the requirements of the algorithm. Even better, we may opt for the same strategy of `SerializableAttribute`, but with more information. This is one idea about how we can design and implement sophisticated engines and advanced business systems. We can define a set of rules like these:

- All designed managed types enabled to take part in a cloning operation *must* have `Business.CloneableTypeAttribute` applied.
- All members designed to compose the state of the instance *and* can return an adequate value to a cloning operation *must* have `Business.CloneableMemberAttribute` applied. `Business.CloneableMemberAttribute` receives as an argument one of the values of an enumeration `Business.CloneableMemberType` that indicates the type of the member (field, property, method, event, and so on).
- We can create a more sophisticated set of attributes and rules based on them, including member access rules, if the value must be different from null and so on.

Now let's try to create a custom experimental implementation of the inherited `Object.MemberwiseClone()` method and the implementation for the `ICloneable` interface in our `Person` custom reference type. First, we create a specialized implementation of `Object.MemberwiseClone()` for our contextualized type model (type hierarchy). As the reference type is a root type, all business derived types, directly or indirectly, will inherit this base implementation. First, we have declared the enumeration `CloneType` with two implementation techniques to the clone operation `CloneType.Shallow` and `CloneType.Deep`, and we can add more implementation techniques as needed. To minimize or even avoid extra public clone method implementations with exotic names, we have the root interface `IBusiness` with one method for that operation, which we assume is the `Clone()` instance method but with

a parameter of the `CloneType` enumeration. This is the root interface for every business object and must have standard operations supported by any business type, including the clone operation within any type in the business system, derived or not, from the `Person` custom type. The implementation of `IBusiness` must follow at least these two rules: the method must be an explicit interface implementation and must be declared with the `public` access member modifier. The other two are what I had described. `Object.MemberwiseClone()` and `ICloneable` should be public and must be used as support methods to specialized cloning operations, like this experimental implementation used on this example. These are especially important design and implementation rules.

In Listing 3-47 we have the enum `CloneType` that is used to identify the type of clone operation and the `IBusiness` generic interface to the clone operation that our business types must implement if they need a custom clone operation implementation.

Listing 3-47. CloneType enum and IBusiness Generic Interface with Clone Operation Generic Method (Implementation Using C++/CLI)

```
public enum CloneType {

      Shallow,
      Deep
};

generic<typename TBusinessType>

interface class IBusiness {

      /* The method is implicitly virtual abstract. */

      TBusinessType Clone( CloneType operationType );
};
```

In Listing 3-48 we have the implementation of clone operations to the `Person` custom reference type. We must override the `MemberwiseClone()` method inherited from `System.Object` and implement the `ICloneable.Clone()` method using our business rules. The other specialized clone implementation is to our `IBusiness` generic interface.

Listing 3-48. Custom Experimental Implementation for the Clone Operation (Implementation Using C++/CLI)

```
public ref class Person: System::ICloneable, public ::IBusiness<Person^> {

      protected:
            virtual Object ^ MemberwiseClone() new {
               /* Copy the current state for the clone. */
                  return { gcnew Person( this->Name, this->Age ) };
            };

            virtual Object^ Clone() = ICloneable::Clone{
                  return this->MemberwiseClone();
            };

      public:
            virtual Person^ Clone( CloneType operationType ) =
            IBusiness<Person^>::Clone{
                  return ( ( operationType == CloneType::Shallow ) ? safe_
                  cast<Person^>( ( (ICloneable^) this )->Clone() ) :
                  nullptr );
{ /* Replace nullptr by an internal implementation for a deep clone
operation. */ }
            };
```

In Listing 3-49 and Listing 3-50 we have the same custom implementation but using the C# programming language.

Listing 3-49. CloneType Enum and IBusiness Generic Interface with the Clone Operation Generic Method (Implementation Using C#)

```
public enum CloneType {

      Shallow,
      Deep
};

interface IBusiness<TBusinessType> {
      TBusinessType Clone( CloneType operationType );
};
```

Listing 3-50. Custom Experimental Implementation for the Clone Operation (Implementation Using C#)

```
public class Person : System.ICloneable, IBusiness<Person> {

      protected new Object MemberwiseClone() {

            /* Copy the current state for the clone. */

            return new Person( this.Name, this.Age );

      }

      Object ICloneable.Clone() {

            return this.MemberwiseClone();

      }

      Person IBusiness<Person>.Clone( CloneType operationType ) {

            return ( ( operationType == CloneType.Shallow ) ? ( ( (
            ICloneable ) this ).Clone() as Person ) :
            null );
            { /* Replace null by an internal implementation for a deep
            clone operation. */ }

      }
```

In the implementation with C++/CL, in terms of technical possibilities, the `IBusiness` interface can be declared as a template or be generic, as shown in Listing 3-51. When declared as a template, it is resolved and compiled at compile time, and the generic implementation is instantiated at runtime by the VES, which means that the CIL generated is specific for use with that assembly.

Listing 3-51. Declaration of the IBusiness Interface

```
generic<typename TBusinessType>
//template<typename TBusinessType>
interface class IBusiness {

      /* The method is implicitly virtual abstract. */

      TBusinessType Clone( CloneType operationType );

};
```

Regarding the metadata in CLI, there are no encodings, data structures, and symbolic values for explicitly identifying the template, and the only fundamental rule is that the emitted name for any template class must not be written in a CLS-compliant form. As shown in Listing 3-52, the name for the template class is `IBusiness<Person^>`, and as shown in Listing 3-53, the name for the generic class is `!0 class Business`1<class Person>` using the arity notation `` `n `` specified in the CLI specification.

Listing 3-52. CIL Emitted from Template Declaration and Implementation

```
.method public hidebysig newslot virtual
        instance class Person  Clone(valuetype CloneType operationType) cil
        managed
{
  .override 'IBusiness<Person ^>'::Clone
  // Code size       20 (0x14)
  .maxstack  1
  .locals ([0] class Person V_0)
  IL_0000:  ldarg.1
  IL_0001:  brtrue.s   IL_0010
  IL_0003:  ldarg.0
  IL_0004:  callvirt   instance object [mscorlib]System.ICloneable::Clone()
  IL_0009:  castclass  Person
  IL_000e:  br.s       IL_0011
  IL_0010:  ldnull
  IL_0011:  stloc.0
  IL_0012:  ldloc.0
  IL_0013:  ret
} // end of method Person::Clone
```

Listing 3-53. CIL Emitted from Generic Declaration and Implementation

```
.method public hidebysig newslot virtual
        instance class Person  Clone(valuetype CloneType operationType) cil
        managed
{
  .override  method instance !0 class IBusiness`1<class
Person>::Clone(valuetype CloneType)
```

```
  // Code size       20 (0x14)
  .maxstack  1
  .locals ([0] class Person V_0)
  IL_0000:  ldarg.1
  IL_0001:  brtrue.s   IL_0010
  IL_0003:  ldarg.0
  IL_0004:  callvirt   instance object [mscorlib]System.ICloneable::Clone()
  IL_0009:  castclass  Person
  IL_000e:  br.s       IL_0011
  IL_0010:  ldnull
  IL_0011:  stloc.0
  IL_0012:  ldloc.0
  IL_0013:  ret
} // end of method Person::Clone
```

When a client code tries to use these experimental implementations, we have the following two scenarios because `IBusiness.Clone()` is an explicit interface implementation:

- *Scenario 1*: In C++/CLI client code we can use the typical call or use case. This is shown in Listing 3-54.
- *Scenario 2*: In C# client code we cannot call `IBusiness.Clone()` directly; we must use casting. This is shown in Listing 3-55.

Listing 3-54. Using the Customized Experimental Implementation of the Clone Operation with C++/CLI Client Code

```
/* Works. */
( ( IBusiness<Person^>^ ) prsA )->Clone( CloneType::Shallow );

/* Works. */
Person^ myClone{ prsA->Clone( CloneType::Shallow ) };
```

Listing 3-55. Using the Customized Experimental Implementation of Clone Operation with C# Client Code

```
/* Works. */
( ( IBusiness<Person> ) prsA ).Clone( CloneType.Shallow );

/* Does not work. */
prsA.Clone( CloneType.Shallow );
```

When considering the creation of implementation rules, it is important to define usage rules for application code too. For example, based on that experimental implementation, all members based on the explicit interface implementation of `IBusiness` must be used accordingly. That means using an explicit cast to the interface type to avoid shortcuts created by any compiler, linker, programming language, projection, or supposed optimization by any compiler and linker option. If we identify or learn about some optimization based on the programming language or projection, compiler, or linker, we must document the possibilities based on specific scenarios and explain when to use these specific optimization forms and when to use the standard implementation behaviors.

CHAPTER 4

Programming with the Common Intermediate Language

In this chapter, we will learn more details about the CTS and the VES, that is, more about how the execution environment deals with the types and structural elements of the platform. The chapter will use code written directly in CIL to explain more details about the use of types and cover internal aspects of the execution environment.

About the Sample Code

This chapter made extensively use of the sample code to illustrate the concepts covered. We can find the source code in Apress GitHub repository: https://github.com/Apress/pro-dot-net-framework-base-class-library/Code/Ch04/CIL.

To compile the sample code written in CIL, open one of the command prompts configured by Microsoft Visual Studio and use one of these commands:

- `ilasm FileName.il`
- `ildasm /X64 FileName.il` (creates a 64-bit image)

If we want to see the generated image via `ildasm.exe`, just use this command:

```
ildasm FileName.exe
```


R. Villela, *Pro .NET Framework with the Base Class Library*, https://doi.org/10.1007/978-1-4842-4191-2_4

Virtual Execution System

The responsibility of the execution system is to provide an environment for the execution of managed code. In fact, the fundamental purpose of the VES described in the CLI specification is to provide the support for the execution of the CIL set. So, it is not possible to talk about the VES and not talk about the CIL set.

The implementation designed and described by the CLI uses an evaluation stack. The intermediate instructions that copy values from memory to the evaluation stack are the *load instructions*. The intermediate instructions that copy values from the evaluation stack back to memory are the *store instructions*.

In Listing 4-1, `ldc.i4.s`, `ldloc.s`, `ldloca.s`, and `ldstr` are examples of load instructions, and `stloc.s` is an example of a store instruction. Each of the CIL instructions is identified with one unique operation code (*opcode*).

Unique opcodes can be 1 byte or 2 bytes in size. When the operation code is 2 bytes, the first byte of the code is always the value `0xFE` (hexadecimal notation). Here are the opcodes of important instructions used in Listing 4-1:

- 2-byte instruction
  - `ceq` (compare equal) with the opcode value `0xFE01`
- 1-byte instructions
  - `br.s` (unconditional branch/short-parameter form) with the opcode value `0x2B`
  - `brfalse.s` (branch on `false`, `null`, or zero) with the opcode value `0x2C`
  - `ldc.i4.s` (load numeric constant/short-parameter form) with the opcode value `0x1F`
  - `ldloc.s` (load local variable onto the stack/short-parameter form) with the opcode value `0x11`
  - `ldloca.s` (load local variable address/short-parameter form) with the opcode value `0x12`
  - `ldstr` (load a literal string) with the opcode value `0x72`
  - `stloc.s` (pop the value from the stack to local variable/short-parameter form) with the opcode `0x13`

Following each opcode is the byte stream with the arguments for the parameters, if the instruction has any.

Listing 4-1. CIL of the File Example_Ch_04_00.il, Load and Store Operations

```
/*

Author: Roger Villela

CIL - Common Intermediate Language Instructions.

Demonstration

      Load instructions and Store instructions.

*/

.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch04 {
}

.method static public void MyEntryPointMethod() cil managed {

.entrypoint

.locals init (
      int32 vltA,
      int32 vltB,
      bool isEqual,
      bool result
)

ldc.i4.s           0x64  /* Load constant. */
stloc.s            vltA  /* Store. */

ldloc.s            vltA
stloc.s            vltB

/* Load the values in the stack for comparison operation. */
ldloc.s    vltA
ldloc.s    vltB
```

```
ceq

stloc.s    isEqual
ldloc.s    isEqual

brfalse.s  IL_00            /* Branch on false. */

ldstr      "vltA == vltB: {0}"
ldloca.s   isEqual
call       instance string [mscorlib]System.Boolean::ToString()
call       void [mscorlib]System.Console::WriteLine(string,  object)

br.s       IL_Final         /* Branch unconditionally. */

IL_00:

ldstr      "Bla, Bla, Bla..."
call       void [mscorlib]System.Console::WriteLine(string)

IL_Final:

call       string [mscorlib]System.Console::ReadLine()
pop

ret

}
```

Now we start learning about the fundamental behaviors of the execution environment using operations available on the CIL set.

Using the Stack

As we have read, the load instructions push items onto the top of the stack, and the store instructions pop out items from the top of the stack and store them to a memory location, represented by a variable, for example. But there are other specialized instructions for push items on the stack, as listed here:

- 1-byte instructions
  - `dup` (duplicate the top value of the stack) with the opcode value `0x25`
  - `pop` (remove the top element of the stack) with the opcode value `0x26`

Listing 4-2 shows the use of dup, and Listing 4-3 shows the use of the pop CIL instruction.

Listing 4-2. CIL for the File Example_Ch_04_01.il

```
/*

Author: Roger Villela

CIL - Common Intermediate Language Instructions.

Demonstration

      dup (duplicate the top value of the stack) with the operation code
      (opcode) value 0x25.

*/

.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch04 {
}

.method static public void MyEntryPointMethod() cil managed {

.entrypoint

.locals init (
      bool isEqual
)

/* Put onto the stack the values for comparison. */

ldc.i4.s            0x72  /* Load constant. */

dup

ceq

stloc.s    isEqual /* Remove from top of the evaluation stack. */

ldstr      "Result: {0}"
ldloca.s   isEqual
call       instance string [mscorlib]System.Boolean::ToString()
```

```
call       void [mscorlib]System.Console::WriteLine(string,  object)

call       string [mscorlib]System.Console::ReadLine()
pop

ret

}
```

Listing 4-3. CIL for the File Example_Ch_04_02.il

```
/*

Author: Roger Villela

CIL - Common Intermediate Language Instructions.

Demonstration

      pop (remove the top element of the stack) with the operation code
      (opcode) value 0x26.

*/

.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch04 {
}

.method static public void MyEntryPointMethod() cil managed {

.entrypoint

.locals init (
      int32 index
)

/* The value to be loaded is specified by the operation code itself.  */

/*
ldc.i4.8
ldc.i4.7
ldc.i4.6
ldc.i4.5
```

```
ldc.i4.4
ldc.i4.3
ldc.i4.2
ldc.i4.1
ldc.i4.0
*/
call    void [mscorlib]System.Console::Clear()
ldstr       "Constant value: {0}"
ldc.i4.6

/* Using the pop instruction, remove the top item from the stack and that
is all, the item is not more available for any operation.
If we use this sequence the code print 0 (zero) value. This is the value
that the runtime used to initialize index value type. */

pop
//stloc.s index
ldloca.s index

/* If we uses this sequence, the runtime throws the System.
InvalidProgramException. */

pop
stloc.s index
ldloca.s index

/* If we uses this sequence, the program works and print the value assigned
to the index value type. */

//pop
stloc.s index
ldloca.s index

/* The common use for the pop instruction is to discard non-used top item
of the stack. */

*/
//pop
stloc.s index
```

```
ldloca.s index
call instance string [mscorlib]System.Int32::ToString()

call       void [mscorlib]System.Console::WriteLine(string,  object)

ldstr "\n\nPress <ENTER> to finish..."
call void [mscorlib]System.Console::WriteLine( string )
call    string [mscorlib]System.Console::ReadLine()
pop

ret

}
```

The .locals Directive and the init Keyword

The purpose of the `.locals` directive is to declare local variables. If the `init` keyword is specified, the memory allocated and associated with the local variable must be zeroed for value types and must receive a `null` value when there are variables for object reference types.

Listing 4-4 shows an example of using the `.locals` directive.

Listing 4-4. CIL for the File Example_Ch_04_03.il

```
/*

Author: Roger Villela

CIL - Common Intermediate Language Instructions.

Demonstration

      .locals init (declare local variable and requires that be initialized
      by execution environment).

*/

.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch04 {
}
```

```
.method static public void MyEntryPointMethod() cil managed {
.entrypoint

.locals init (
      int32 valueOne, string message
)

ldstr      "valueOne: {0}"
ldloca.s   valueOne
call       instance string [mscorlib]System.Int32::ToString()
call       void [mscorlib]System.Console::WriteLine(string,  object)

ldstr      "message: {0}"
ldloc.s message

/* Using this block, show non-null value message */
/*
ldstr "Test"
ldloc.s message
stloc.s message
*/

brfalse.s IL_Null_Message
ldstr "non-null value."
br.s IL_Not_Null
IL_Null_Message:
ldstr "null value."

IL_Not_Null:
call       void [mscorlib]System.Console::WriteLine(string,  object)

call       string [mscorlib]System.Console::ReadLine()
pop

ret

}
```

Loading Constant Values

The instructions for constant loading have just one parameter and load it on the evaluation stack. We must explicitly specify the value and cannot use a local variable or some parameter name and the respective argument. We can use decimal or hexadecimal notation.

Listing 4-5 shows the use of instructions to load constants, and Listing 4-6 shows specialized instructions for load constants that do not have parameters, because the value is specified in the opcode.

Listing 4-5. CIL for the File Example_Ch_04_04.il

```
/*

Author: Roger Villela

CIL - Common Intermediate Language Instructions.

Demonstration

      .ldc.<type> (load numeric constant) with the operation code (opcode)
      value 0x20.

*/

.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch04 {
}

.method static public void MyEntryPointMethod() cil managed {
.entrypoint

.locals init ( int32 number )

ldstr        "Constant value: {0}"
ldc.i4 0x72
stloc.s number
ldloca.s number
call       instance string [mscorlib]System.Int32::ToString()
call       void [mscorlib]System.Console::WriteLine(string,  object)
```

```
call       string [mscorlib]System.Console::ReadLine()
pop

ret}
```

Listing 4-6. CIL for the File Example_Ch_04_05.il

```
/*

Author: Roger Villela

CIL - Common Intermediate Language Instructions.

Demonstration

      .ldc.i4.0 (load numeric constant 0) with the operation code (opcode)
      value 0x16.
      .ldc.i4.1 (load numeric constant 1) with the operation code (opcode)
      value 0x17.
      .ldc.i4.2 (load numeric constant 2) with the operation code (opcode)
      value 0x18.
      .ldc.i4.3 (load numeric constant 3) with the operation code (opcode)
      value 0x19.
      .ldc.i4.4 (load numeric constant 4) with the operation code (opcode)
      value 0x1A.
      .ldc.i4.5 (load numeric constant 5) with the operation code (opcode)
      value 0x1B.
      .ldc.i4.6 (load numeric constant 6) with the operation code (opcode)
      value 0x1C.
      .ldc.i4.7 (load numeric constant 7) with the operation code (opcode)
      value 0x1D.
      .ldc.i4.8 (load numeric constant 8) with the operation code (opcode)
      value 0x1E.
      .ldc.i4.m1 (load numeric constant -1) with the operation code
      (opcode) value 0x15.
      .ldc.i4.M1 (load numeric constant -1) with the operation code
      (opcode) value 0x15.

      The .ldc.i4.M1 is an alias for the .ldc.i4.m1 instruction.
```

```
*/

.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch04 {
}

.method static public void MyEntryPointMethod() cil managed {
.entrypoint

.locals init ( int32 number )

call       void [mscorlib]System.Console::Clear()

ldstr        "\n\nConstant value: {0}"
ldc.i4.0
stloc.s number
ldloca.s number
call       instance string [mscorlib]System.Int32::ToString()
call       void [mscorlib]System.Console::WriteLine(string,  object)

ldstr "\n\nPress<ENTER> to finish..."
call       void [mscorlib]System.Console::WriteLine(string)
call       string [mscorlib]System.Console::ReadLine()
pop

ret

}
```

Working with Vector Arrays

An instance of `System.Array` that is a vector array (unidimensional array) is a special type. So, how do we work with it?

Here are some instructions and the opcodes, and we can read details about these instructions by visiting the documentation at https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes?redirectedfrom=MSDN&view=netframework-4.7.2#fields.

- `newarr` (compare equal) with the opcode value `0x8D`
- `ldelem.i1` (load a vector element with type `int8`) with the opcode value `0x90`
- `ldelem.u1` (load a vector element with type unsigned `int8`) with the opcode value `0x91`
- `ldelem.i2` (load a vector element with type `int16`) with the opcode value `0x92`
- `ldelem.u2` (load a vector element with type unsigned `int16`) with the opcode value `0x93`
- `ldelem.i4` (load a vector element with type `int32`) with the opcode value `0x94`
- `ldelem.u4` (load a vector element with type unsigned `int32`) with the opcode value `0x95`
- `ldelem.i8` (load a vector element with type `int64`) with the opcode value `0x96`
- `stelem.i1` (store a value in a vector element of type `int8`) with the opcode value `0x9C`
- `stelem.i2` (store a value in a vector element of type `int16`) with the opcode value `0x9D`
- `stelem.i4` (store a value in a vector element of type `int32`) with the opcode value `0x9E`
- `stelem.i8` (store a value in a vector element of type `int64`) with the opcode value `0x9F`

In the CIL instruction set there are specific instructions for dealing with vector arrays but not with nonvector arrays. So we can more easily understand this, the listings are organized by the following operations: creating a vector, loading an element of the vector, and storing an element on the vector.

Listing 4-7 shows the instruction for creating the vector, Listing 4-8 shows the instruction for querying the vector length, Listing 4-9 shows the instruction for loading an element of a vector, and Listing 4-10 shows the instruction for storing an element in a vector.

Listing 4-7. CIL of the File vector_creation.il, Vector Creation

```
/*

Author: Roger Villela

CIL - Common Intermediate Language Instructions.

Demonstration

      newarr (create a zero-based, one-dimensional array) with the
      operation code (opcode) value 0x8D.

*/

.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch04 {
}

.method static public void MyEntryPointMethod() cil managed {

.entrypoint

/* Declare the vector and initialize the associated allocated block of
managed memory . */

.locals init ( int32[] numbers )

/* Load constant with the number of items for the new vector. */
ldc.i4.s            0xA

/* Vector creation. */
newarr [mscorlib] System.Int32
stloc.0

call      void [mscorlib]System.Console::Clear()
ldstr "\n\n\nVectors number created.\n\n\n"
call       void [mscorlib]System.Console::WriteLine(object)

ldstr "Press <ENTER> to finish..."
call       void [mscorlib]System.Console::WriteLine(object)
call       string [mscorlib]System.Console::ReadLine()
```

```
pop

ret

}
```

Listing 4-8. CIL of the File vector_length.il, Vector Length

```
/*

Author: Roger Villela

CIL - Common Intermediate Language Instructions.

Demonstration

.ldlen (load the length of an array) with the operation code (opcode) value 0x8E.

*/

.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch04 {
}

.method static public void MyEntryPointMethod() cil managed {

.entrypoint

/* Declare the vector and initialize the associated allocated block of
managed memory . */

.locals init ( int32[] numbers, int32 length )

/* Load constant with the number of items for the new vector. */
ldc.i4.s            0xA

/* Vector creeation. */
newarr [mscorlib] System.Int32
stloc.0

call      void [mscorlib]System.Console::Clear()
```

```
ldstr "\n\n\nVector numbers created.\n\n\n"
call       void [mscorlib]System.Console::WriteLine(object)
ldloc.0

/* Get the length (number of elements) of array. */

ldlen
stloc.1

ldstr "\n\n\nVector length: {0}.\n\n\n"
ldloca.s length
call instance string [mscorlib] System.Int32::ToString()
call       void [mscorlib]System.Console::WriteLine(string, object)

ldstr "\n\n\nPress <ENTER> to finish..."
call       void [mscorlib]System.Console::WriteLine(object)
call      string [mscorlib]System.Console::ReadLine()
pop

ret}
```

Listing 4-9. CIL of the File vector_length.il, Loading an Element of the Vector

```
/*

Author: Roger Villela

CIL - Common Intermediate Language Instructions.

Demonstration

      ldelem.i4 (Load a vector element of type int32) with the operation
      code (opcode) value 0x94.

*/

.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch04 {
}

.method static public void MyEntryPointMethod() cil managed {

.entrypoint
```

```
/* Declare the vector and initialize the associated allocated block of
managed memory . */
.locals init ( int32[] numbers, int32 index, int32 item )

/* Load constant with the number of items for the new vector. */
ldc.i4.s            0xA

/* Vector creation. */
newarr [mscorlib] System.Int32
stloc.0

call      void [mscorlib]System.Console::Clear()

ldstr "\nVector numbers created.\n"
call       void [mscorlib]System.Console::WriteLine(object)
ldloc.0
ldlen
stloc.1
ldstr "\nVector length: {0}.\n"
ldloca.s index
call instance string [mscorlib] System.Int32::ToString()
call       void [mscorlib]System.Console::WriteLine(string, object)

ldstr "\nVector item {0} at index {1}.\n"

ldloc.0
/* Third position. */
ldc.i4.s 0x2
ldelem.i4
stloc.2
ldloca.s item
call instance string [mscorlib] System.Int32::ToString()

/* Third position. */
ldc.i4.s 0x2
stloc.1
ldloca.s index
call instance string [mscorlib] System.Int32::ToString()
```

```
call       void [mscorlib]System.Console::WriteLine(string, object, object)

ldstr "\n\n\nPress <ENTER> to finish..."
call       void [mscorlib]System.Console::WriteLine(object)
call      string [mscorlib]System.Console::ReadLine()
pop

ret

}
```

Listing 4-10. CIL of the File vector_element_storing.il, Vector Storing

```
/*

Author: Roger Villela

CIL - Common Intermediate Language Instructions.

Demonstration

      stelem.i4 (Store a value in a vector element of type int32) with the
      operation code (opcode) value 0x9E.

*/

.assembly extern mscorlib {}

.assembly ProDotNETBCL.Vol1.Ch04 {
}

.method static public void MyEntryPointMethod() cil managed {

.entrypoint

/* Declare the vector and initialize the associated allocated block of
managed memory . */

.locals init ( int32[] numbers, int32 index, int32 item )

/* Load constant with the number of items for the new vector. */
ldc.i4.s            0xA
```

```
/* Vector creation. */
newarr [mscorlib] System.Int32
stloc.0

call   void [mscorlib]System.Console::Clear()
ldstr "\nVector numbers created.\n"
call   void [mscorlib]System.Console::WriteLine(object)

ldloc.0

/* Third position. */
ldc.i4.s            0x2
stloc.1
ldloc.1

/* Value to store. */
ldc.i4.s            0x72
stloc.2
ldloc.2

/* Element storing. */
stelem.i4

ldloc.0
ldlen
stloc.1
ldstr "\nVector length: {0}.\n"
ldloca.s index
call instance string [mscorlib] System.Int32::ToString()
call    void [mscorlib]System.Console::WriteLine(string, object)

ldstr "\nVector item {0} at index {1}.\n"

ldloc.0

/* Third position. */
ldc.i4.s 0x2

/* Element loading. */
ldelem.i4
```

```
stloc.2
ldloca.s item
call instance string [mscorlib] System.Int32::ToString()

/* Third position. */
ldc.i4.s 0x2
stloc.1
ldloca.s index
call instance string [mscorlib] System.Int32::ToString()

call    void [mscorlib]System.Console::WriteLine(string, object, object)

ldstr "\n\n\nPress <ENTER> to finish..."
call      void [mscorlib]System.Console::WriteLine(object)
call      string [mscorlib]System.Console::ReadLine()
pop

ret


}
```

In this chapter, we learned that the VES is responsible for not only executing instructions but for guaranteeing that the CIL instructions will be executed correctly. It is important to experiment with a subset of fundamental operations, such as storing and loading, via code examples. The purpose of this chapter was to cover the more practical aspects of the relationship between the CTS and the VES.

In the next chapter, we will continue our exploration of the other aspects of the execution environment using the CIL instruction set, but including the structural and representational element of metadata.

CHAPTER 5

Assembly Manifest and Versioning

This chapter will cover the metadata system and its importance to the CTS and the VES. For this, we use one of the critical structural and behavioral elements of the platform: the assembly manifest. We will also begin learning about the versioning system and how the VES works with it.

Metadata

The basic definition of *metadata* is "data that describes data." However, this is too abstract of a concept on the first read. We can experiment with the benefits of metadata using the `System.Reflection` types and the `System.Type` reference type, for example. As the names of namespaces and types imply, the purpose is to get information about the assemblies, the modules, and the managed types.

The Assembly Manifest

The assembly is specified as a module that holds the manifest in the metadata. Every module should have at least one `.assembly` directive. For example, if we are working in the business of transportation cargo, we know that one of the most critical aspects is guaranteeing that the requested products are correctly dispatched and delivered. One of the tools used to organize these activities is the manifest of cargo. Likewise, when we are working with the .NET Framework libraries, they are organized as products that need cargo manifests so that the execution environment can enforce what is described in the cargo manifests when the application requests a product.

R. Villela, *Pro .NET Framework with the Base Class Library*, https://doi.org/10.1007/978-1-4842-4191-2_5

The metadata is the technological element that makes this possible. Instead of using only configuration files or a proprietary and specialized database configuration system like the Microsoft Windows Registry, using metadata creates a conceptual way to describe structures and their responsibilities, and it a representative adaptable form that is independent of the hardware platform and software platform (operating system). The information that is stored in the modules and created through assemblies is what helps the execution environment understand and apply the rules for the relations among the elements.

There are four elements need to construct a static assembly for any .NET software components and .NET software applications.

- The CIL implements all the managed types and required logic for the .NET software component and software application.
- Metadata.
- The resources (audio/video files, localization support files, images, and custom files created specifically for the application).
- The assembly manifest.

From the perspective of the execution environment and basic structural rules described in the CLI, when we create an assembly, the only required item is the assembly manifest. The assembly manifest holds the assembly metadata. However, when considering even the simplest .NET software components or software application, if we does not have the other elements, then the components or applications will not have a practical use.

Structural Organization of the Assembly Manifest

The organization of the assembly manifest is based on the description and storage of the collection of elements that make up the assembly and how they relate to each other. This information is part of the assembly metadata.

Specifically, the assembly manifest holds the following information:

- The assembly must have version requirements, even the most basic.
- The assembly must have the security identity requirements fulfilled.
- The assembly must have a well-defined scope.
- The assembly must have information about the references for other assemblies. This information helps the execution environment to find these referenced assemblies and enforce the rules.
- The assembly must have information about specialized resources, such as external image files that are part of the assembly; that is, they must be deployed and available when application requires.

The following are two important aspects to remember:

- For the Microsoft Windows operating system, the assembly manifest is stored in a PE file (module) or EXE or DLL, which can also have CIL code.
- For the Microsoft Windows operating system, the assembly manifest can be stored in a PE file (module) with just the assembly manifest and nothing more.

As shown in Listing 5-1, using the example written in CIL, when we are creating the static assembly with just one file module associated, the assembly manifest is automatically incorporated into the file module (PE). This way of organization forms a single-module assembly or single-file assembly. As shown in Figure 5-1, using the ILDasm tool, we can see that the manifest is an item with a distinct appearance in the tool. Figure 5-2 shows details about the manifest.

Listing 5-1. A Single-File Assembly with the Assembly Manifest Incorporated into the Module and CIL Implementing the Logic of the Application

```
/*

Author: Roger Villela

CIL - Common Intermediate Language Instructions.
```

```
Demonstration

      .ldlen (load the length of an array) with the operation code (opcode)
      value 0x8E.

*/

.assembly extern mscorlib {}

.assembly RVJ.Listing_5_0 {
}

.method static public void MyEntryPointMethod() cil managed {

.entrypoint

/* Declare the vector and initialize the associated allocated block of
managed memory . */
.locals init ( int32[] numbers, int32 length )

/* Load constant with the number of items for the new vector. */
ldc.i4.s            0xA

/* Vector creation. */
newarr [mscorlib] System.Int32
stloc.0

call    void [mscorlib]System.Console::Clear()
ldstr "\n\n\nVector numbers created.\n\n\n"
call    void [mscorlib]System.Console::WriteLine(object)
ldloc.0
/* Get the length (number of elements) of array. */
ldlen
stloc.1
ldstr "\n\n\nVector length: {0}.\n\n\n"
ldloca.s length
call instance string [mscorlib] System.Int32.ToString()
call    void [mscorlib]System.Console::WriteLine(string, object)
```

```
ldstr "\n\n\nPress <ENTER> to finish..."
call    void [mscorlib]System.Console::WriteLine(object)
call    string [mscorlib]System.Console::ReadLine()
pop

ret }
```

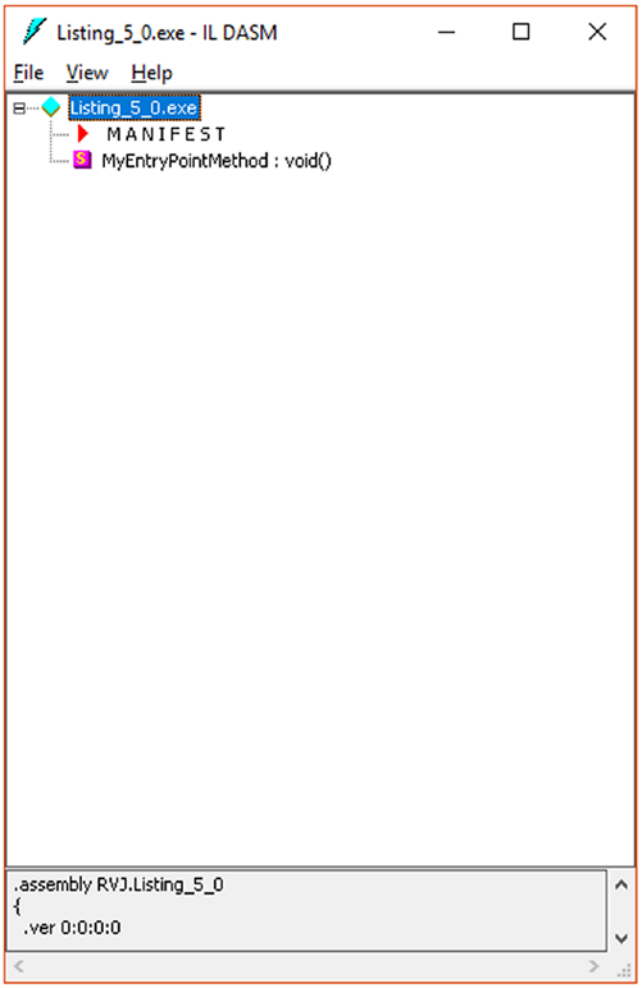


Figure 5-1. *The managed content of the binary shown in the ILDasm tool*

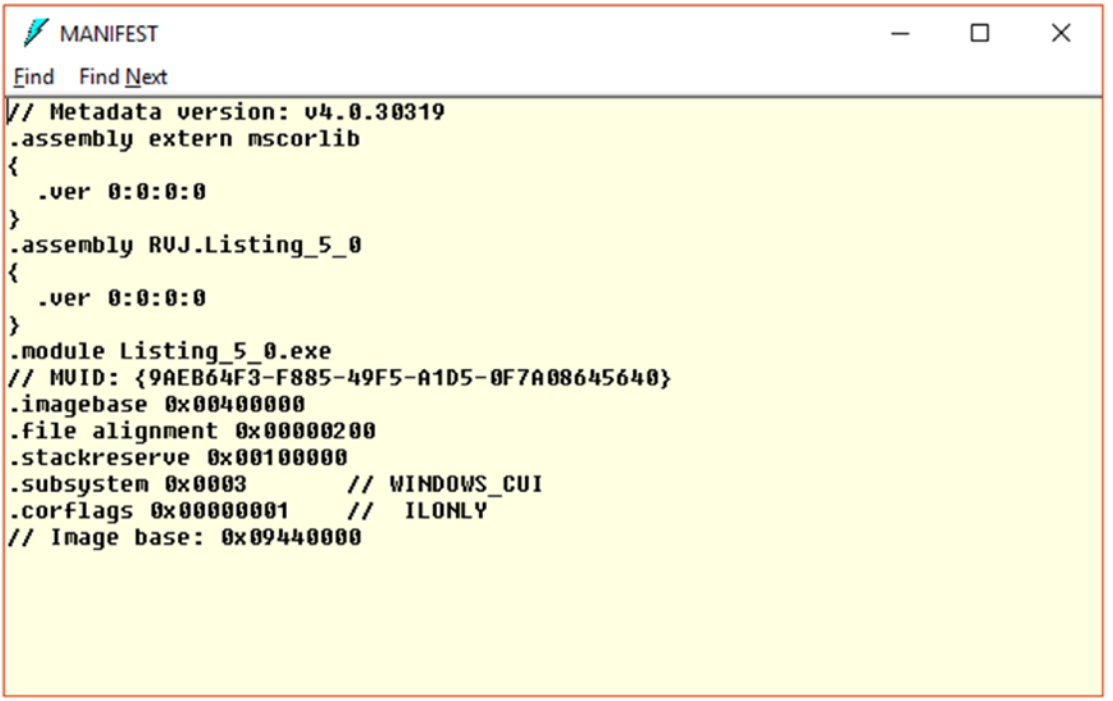


Figure 5-2. *The content of the assembly manifest shown in the ILDasm tool*

In Listing 5-2 we can see this comment:

```
// Metadata version: v4.0.30319
```

This specifies two things: the full version number (`<major>.<minor>.<build>.<revision>`) of the CLR and the metadata stream version supported by that specific implementation of the runtime.

Listing 5-2. Content of the Assembly Manifest

```
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .ver 0:0:0:0
}
.assembly RVJ.Listing_5_0
{
  .ver 0:0:0:0
}
```

```
.module Listing_5_0.exe
// MVID: {9AEB64F3-F885-49F5-A1D5-0F7A08645640}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x09440000
```

Working with the Version Directive and Values

The `.ver` assembly directive sets the assembly version. This information is part of the assembly manifest and is stored in the assembly metadata. This is the value read by the VES to get the assembly version.

As shown in Listing 5-3, when we are using the CIL, we can set the value for the `.ver` assembly directive directly in the intermediate code.

Listing 5-3. The .ver Assembly Directive

```
.assembly extern mscorlib {
.ver 4:0:0:0
}
.assembly RVJ.Listing_5_2 {
.ver 1:0:0:0
}
```

The version number has four parts: `<major version>.<minor version>.<build number>.<revision number>`. The four parts use the `System.Int32` value type. Although the parts use the `System.Int32` value type, the four parts of the version number can use a range from 0 to `System.UInt16.MaxValue` - 1. These supported values should be checked during the compilation process. That is, the compilation tool should check whether the informed version value is in the valid range. When the assembly is loaded, the VES

also checks this version number value but retrieves it from the assembly metadata. The version number 0:0:0:0 is an aspect of the ILasm compiler and is a valid version number. As we can see, the compilation tool does not use 0:0:0:0 as default starting value and instead uses 1:0:0:0 as default start version number. The CLI specifies that an assembly version number should be obtained at compile time and that all assemblies in conformance with the standard specification should have the last two version numbers set with the 0 value. In other words, when not explicitly specifying the last two numbers, `<build number>.<revision number>`, they cannot be random or negative numbers. But considering the valid range of values, the standard specification does not specify a "default" start value that should be used by the compilation tools.

Using System.Reflection.AssemblyVersionAttribute

The value for the `.ver` assembly directive should be obtained and set at compile time, as specified by CLI. To more easily capture this version number value, it is used by the `System.Reflection.AssemblyVersionAttribute` custom attribute. IDE-based products, such as Microsoft Visual C# and Microsoft Visual C++, use custom attributes to capture values based on the information from the project properties and supplied by the development environment itself, according to the project type. As we are learning in this chapter and the book, the core mechanisms of the platform are built around the support for the metadata information and for the execution system. `System.Reflection.AssemblyVersionAttribute` is a reference type for the abstract concept of an attribute. From the CTS perspective, an attribute is a reference type derived directly or indirectly from the `System.Attribute` reference type. The `System.Attribute` reference type is part of the BCL and is in the assemblies `mscorlib` (`mscorlib.dll`), `System.Runtime` (`System.Runtime.dll`), and `netstandard` (`netstandard.dll`).

No system is complete at a specific moment, because all things change all the time. Instead of creating a "closed" system with no perspective for extensibility and aggregation of information, resources, and functionalities, the platform offers the possibility to extend the understanding of context using metadata. As we have learned with the `.ver` assembly directive, the standard specification requires that an assembly version number should be obtained and set at compile time. For example, using CIL if we do not explicitly specify an assembly version number, the ILasm compiler sets the 0:0:0:0 version value for the `.ver` assembly directive. But we can set the assembly version number using the `System.Reflection.AssemblyVersionAttribute` custom attribute too.

In fact, this is the way that most compilations tools capture and set the value for the `.ver` assembly directive.

As shown in Listing 5-4, we can read the declaration of the `System.Reflection.AssemblyVersionAttribute` reference type using the C# programming language and using C++/CLI syntax. In addition, we can read other custom attributes that have been applied to the `System.Reflection.AssemblyVersionAttribute` reference type. The metadata information of these custom attributes is available statically and at runtime can be manipulated and used by any application code that we can write. In fact, custom attributes are used by Microsoft visual designers, by compilers, and by the VES of the platform. As the things are improved all the time, it is important to periodically check the official documentation of `AssemblyVersionAttribute`, available at https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assemblyversionattribute?view=netframework-4.7.2.

Listing 5-4. Declaration of the AssemblyVersionAttribute Reference Type

```
/* C++/CLI (Common Language Infrastructure) projection. */

[AttributeUsageAttribute(AttributeTargets.Assembly, Inherited = false)]
[System.Runtime.InteropServices.ComVisible(true)]
public ref class AssemblyVersionAttribute sealed : public Attribute

/* C# programming language. */

[System.AttributeUsage(System.AttributeTargets.Assembly, Inherited=false)]
[System.Runtime.InteropServices.ComVisible(true)]
public sealed class AssemblyVersionAttribute : Attribute
```

All compilation tools, as we have learned, should obtain and set the assembly version number at compile time, and the last two numbers of the assembly version number should be set to 0 (zero). For the .NET projects of Microsoft Visual Studio, the functionalities of Microsoft Visual C# and Microsoft Visual C++ create a file with the names `AssemblyInfo.cs` and `AssemblyInfo.cpp`, respectively. These files have, at least, the set of assembly custom attributes with the argument values informed automatically by the tools and by the development environment. These values can be set using the project properties window or directly in the `AssemblyInfo.extension` file. These fundamental assembly custom attributes are part of the BCL, but specialized assembly

custom attributes could be provided by specialized tools such as compilation tools and graphical user interface designers of Microsoft Visual Studio.

The C# programming language uses the `System.Reflection.AssemblyVersionAttribute` version value to obtain and set the value for the `.ver` assembly directive.

The C++/CLI projection uses the `System.Reflection.AssemblyVersionAttribute` version value to obtain and set the value for the `.ver` assembly directive. In a .NET template project, Microsoft Visual C++ includes `System.Reflection.AssemblyVersionAttribute` and does not include `System.Reflection.AssemblyFileVersionAttribute`. But we can manually include `System.Reflection.AssemblyFileVersionAttribute` and set the argument values for the appropriate constructor.

As shown in Listing 5-5, even in the CIL declared like this excerpt, what prevails is the version number value set for the `.ver` assembly directive.

Listing 5-5. Even with Different Assembly Version Numbers, the .ver Assembly Directive Value Prevails

```
.assembly RVJ.Listing_5_4 {

/*

Even using the System.Reflection.AssemblyVersionAttribute with value
1.0.0.0 and the .ver assembly directive with the runtime version number
value of 6.0.0.0, prevails the value assigned for .ver assembly directive.

*/
.custom instance void [mscorlib]System.Reflection.AssemblyVersion
Attribute..ctor(string) = ( 01 00 07 31 2E 30 2E 30 2E 30 00 00 )

.ver 6:0:0:0

}
```

This fundamental rule applies to any programming language for the platform. Listing 5-6 shows an example in C# of using version attributes in the `AssemblyInfo.cs` file of a Windows Forms application. Listing 5-7 shows an example in C++/CLI of using version attributes in the `AssemblyInfo.cpp` file of a .NET Console application.

ABOUT THE ASSEMBLY VERSION NUMBER AND MODULE VERSION NUMBER

The version number of an assembly is stored in the assembly metadata and is one of the elements that makes up the identity of the assembly.

The assembly version number and the module version number do not need to be equal. We can have different values, as shown in this excerpt:

```
/* C# programming language assembly version number and module version number. */

[assembly: AssemblyVersion( "6.0.0.0" )]
[assembly: AssemblyFileVersion( "7.0.0.0" )]

/* C++/CLI (Common Language Infrastructure) projection assembly version
number and module version number. */

[assembly:AssemblyVersionAttribute("6.0")];
[assembly:AssemblyFileVersionAttribute("7.0")];
```

In the CIL the assembly version number can be set by directly writing the value for the `.ver` assembly directive or the `System.Reflection.AssemblyVersionAttribute` custom attribute. But as part of the compilation process, the ILasm compiler does not use the value of `System.Reflection.AssemblyVersionAttribute` to automatically set the value for the `.ver` assembly directive. If the project written in the CIL is considering the use of `System.Reflection.AssemblyVersionAttribute` as the way to capture the assembly version number, for the compilation process it will be necessary to write a custom code that loads this value from the code file, the .NET module, or even a kind of configuration file and apply the version value to the `.ver` assembly directive.

The module version number is set by `System.Reflection.AssemblyFileVersionAttribute`.

Listing 5-6. AssemblyInfo.cs File of a Windows Forms .NET Project with a Set of Assembly Custom Attributes

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
```

```
// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle( "My Windows Forms App" )]
[assembly: AssemblyDescription( "" )]
[assembly: AssemblyConfiguration( "" )]
[assembly: AssemblyCompany( "" )]
[assembly: AssemblyProduct( "My Windows Forms App" )]
[assembly: AssemblyTrademark( "" )]
[assembly: AssemblyCulture( "" )]

// Setting ComVisible to false makes the types in this assembly not visible
// to COM components.  If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible( false )]

// The following GUID is for the ID of the typelib if this project is
exposed to COM
[assembly: Guid( "1a18a40a-bf33-489c-859e-98ac07dd64ec" )]

// Version information for an assembly consists of the following four values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion( "6.0.0.0" )]
[assembly: AssemblyFileVersion( "7.0.0.0" )]
```

Listing 5-7. AssemblyInfo.cpp File of a Console .NET Project with a Set of Assembly Custom Attributes

```
#pragma region Namespaces
using namespace System;
using namespace System.Reflection;
using namespace System.Runtime.CompilerServices;
using namespace System.Runtime.InteropServices;
using namespace System.Security.Permissions;
#pragma endregion

/*
General Information about an assembly is controlled through the following
set of attributes. Change these attribute values to modify the information
associated with an assembly.

*/
[assembly:AssemblyTitleAttribute(L"")];
[assembly:AssemblyDescriptionAttribute(L"RogerVillela Journal")];
[assembly:AssemblyConfigurationAttribute(L"")];
[assembly:AssemblyCompanyAttribute(L"RogerVillela Journal")];
[assembly:AssemblyProductAttribute(L"RogerVillela Journal")];
[assembly:AssemblyCopyrightAttribute(L"Copyright (c)  RogerVillela Journal")];
[assembly:AssemblyTrademarkAttribute(L"RogerVillela Journal")];
[assembly:AssemblyCultureAttribute(L"")];

/*

Version information for an assembly consists of the following four values:

     Major Version
     Minor Version
     Build Number
     Revision
```

```
You can specify all the value or you can default the Revision and Build
Numbers by using the '*' as shown below:

*/

[assembly:AssemblyVersionAttribute("6.0")];
[assembly:AssemblyFileVersionAttribute("7.0")];

[assembly:ComVisible(false)];

[assembly:CLSCompliantAttribute(true)];
```

Versioning with Different Programming Languages

Independently of the programming language being used, the fundamental way of working with versioning is the same. The differences are that if we are using the tools for the CIL, we can have the tasks organized in more steps, depending on the granularity that we have chosen for our project files and code files. Using Microsoft Visual Studio, these tasks are implicitly made by functionalities of the IDE itself in partnership with Microsoft Visual C++ and Microsoft Visual C#. The parameter values based on the project configuration files and type of project (Universal Windows Platform, Windows Presentation Foundation, Windows Forms, Console, Class Library, ASP.NET, etc.) also are used as part of the arguments for the tool's or IDE's tasks.

As explained, when using the CIL, we use the `.ver` assembly directive and set the value directly in the intermediate code, as shown in Listing 5-8. Even with the CIL we can use the `System.Reflection.AssemblyVersionAttribute` reference type, but the syntax is a little complex for those who do not use the CIL regularly. The use of custom attributes should be declared in the intermediate code using the `.custom` assembly directive. According to the signature of the constructors of the custom attribute, we should inform the parameters list with the respective types of each parameter and the argument values for each of the corresponding parameters. The argument values should be informed as a sequence of bytes between parentheses. The mechanism of the custom attributes is used to store information in the metadata at compile time and to provide access (via reflection) to this metadata information at runtime.

Listing 5-8 shows the `System.Reflection.AssemblyVersionAttribute` attribute used to set the assembly version number with a value of `1.0.0.0`. The constructor has just one parameter, and the parameters are of the `System.String` reference type, which is one of the fundamental built-in types in the CTS and the BCL. In Listing 5-9 we can see the declaration of the `AssemblyVersionAttribute()` construct that receives an argument of the `string` type.

Listing 5-8. Using AssemblyVersionAttribute for the Assembly Version Number with the CIL

```
.assembly extern mscorlib {

.ver 4:0:0:0

}

.assembly RVJ.Listing_5_7 {

/* value 1.0.0.0 */
.custom instance void [mscorlib]System.Reflection.AssemblyVersion
Attribute::.ctor(string) = ( 01 00 07 31 2E 30 2E 30 2E 30 00 00 )

.ver 0:0:0:0

}
```

Listing 5-9. Declaration of the Constructor AssemblyVersionAttribute

```
/* Constructor declaration in C++/CLI. */

public:
AssemblyVersionAttribute( String^ version );

/* Constructor declaration in C# programming language. */

public AssemblyVersionAttribute( String version )
```

As shown in Listing 5-8 and in Figure 5-3, when using the CIL and `System.Reflection.AssemblyVersionAttribute`, the ILasm compiler does not automatically use the argument value of the custom assembly version attribute to set the value for the `.ver` assembly directive. Despite this, the assembly version is correctly set when the VES loads the assembly using the informed version number value in the `.ver` assembly directive.

Figure 5-3. *The assembly manifest with the .ver assembly directive, and the System.Reflection.AssemblyVersionAttribute custom attribute*

Figure 5-3 shows the assembly manifest with the `.ver` assembly directive with the version number 0.0.0.0, as well as `System.Reflection.AssemblyVersionAttribute` with the version number 1.0.0.0.

As shown in Figure 5-4, when we create a project based on one of the Microsoft Visual C# templates like Windows Forms, a file with the name `AssemblyInfo.cs` is created. Listing 5-10 shows the default content of `AssemblyInfo.cs`. As shown in Figure 5-5, the same is true for a .NET project created with Microsoft Visual C++ when using one of the templates for the platform. That is, a file called `AssemblyInfo.cpp` is created. Listing 5-11 shows the default content of `AssemblyInfo.cs`.

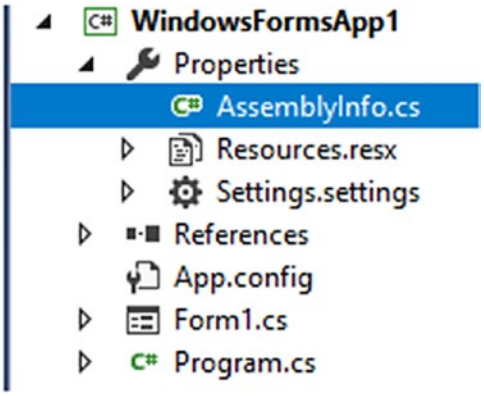


Figure 5-4. *The AssemblyInfo.cs code file automatically created by Microsoft Visual C#*

As we can see in Listing 5-10, the `System.Reflection.AssemblyVersionAttribute` reference type is used to set the assembly version for the project using the C# programming language. The file `AssemblyInfo.cs` is created automatically by Microsoft Visual C#.

Listing 5-10. Content of AssemblyInfo.cs for a Windows Forms Project

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle( "My Windows Forms App" )]
[assembly: AssemblyDescription( "" )]
[assembly: AssemblyConfiguration( "" )]
[assembly: AssemblyCompany( "" )]
[assembly: AssemblyProduct( " My Windows Forms App" )]
[assembly: AssemblyCopyright( "Copyright ©  2018" )]
[assembly: AssemblyTrademark( "" )]
[assembly: AssemblyCulture( "" )]
```

```
// Setting ComVisible to false makes the types in this assembly not visible
// to COM components.  If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible( false )]

// The following GUID is for the ID of the typelib if this project is
exposed to COM
[assembly: Guid( "1a18a40a-bf33-489c-859e-98ac07dd64ec" )]

// Version information for an assembly consists of the following four values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the values or you can default the Build and Revision
Numbers
// by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion( "1.0.0.0" )]
[assembly: AssemblyFileVersion( "1.0.0.0" )]
```

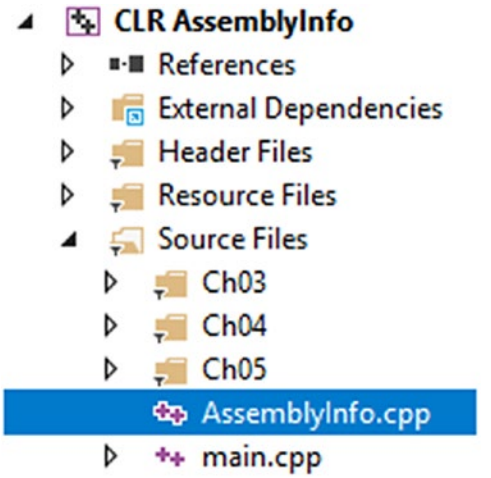


Figure 5-5. *The AssemblyInfo.cpp code file automatically created by Microsoft Visual C++*

Listing 5-11. Content of AssemblyInfo.cpp for a Console Project

```
#pragma region Namespaces
using namespace System;
using namespace System.Reflection;
using namespace System.Runtime.CompilerServices;
using namespace System.Runtime.InteropServices;
using namespace System.Security.Permissions;
#pragma endregion

/*
General Information about an assembly is controlled through the following
set of attributes. Change these attribute values to modify the information
associated with an assembly.

*/
[assembly:AssemblyTitleAttribute(L"")];
[assembly:AssemblyDescriptionAttribute(L"RogerVillela Journal")];
[assembly:AssemblyConfigurationAttribute(L"")];
[assembly:AssemblyCompanyAttribute(L"RogerVillela Journal")];
[assembly:AssemblyProductAttribute(L"RogerVillela Journal")];
[assembly:AssemblyCopyrightAttribute(L"Copyright (c)  RogerVillela Journal")];
[assembly:AssemblyTrademarkAttribute(L"RogerVillela Journal")];
[assembly:AssemblyCultureAttribute(L"")];

/*

Version information for an assembly consists of the following four values:

      Major Version
      Minor Version
      Build Number
      Revision

You can specify all the value or you can default the Revision and Build
Numbers
by using the '*' as shown below:

*/
```

```
[assembly:AssemblyVersionAttribute("1.0")];

[assembly:ComVisible(false)];

[assembly:CLSCompliantAttribute(true)];
```

The CLR Versioning Policy

The CLR has the concept of *policy* as part of the architecture and implementation engineering, and one of these policies is the version policy. Every policy has default configurations known as the *default policy*. Even when we do not explicitly specify a policy, the environment has a set of default configurations present. In a typical implementation of the CLR, one of the rules of the default version policy says that a managed application runs only with the version that it has been built for.

To better understand this, we will use a tutorial. Follow these steps to create a temporary Windows Forms app using Microsoft Visual C#:

1. In Visual Studio, in the menu bar, select File ➤ New ➤ Project and then select Windows Forms Application, as shown in Figure 5-6.

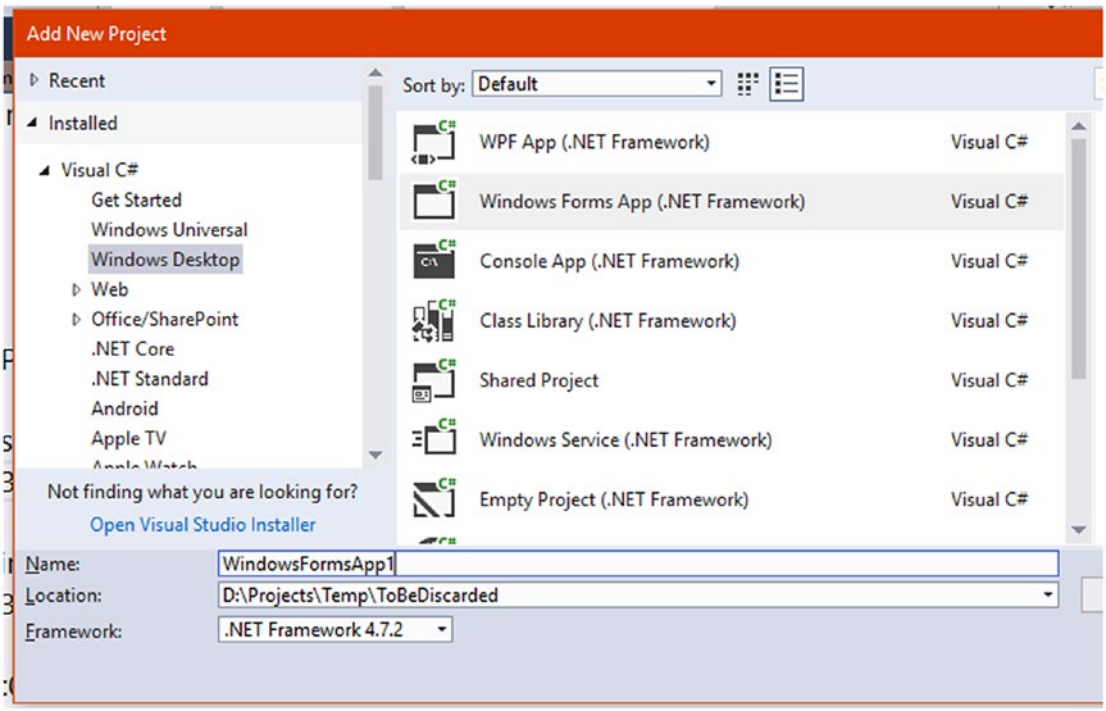


Figure 5-6. *Microsoft Visual C# dialog box for choosing the project type and the .NET Framework version for the project*

2. As shown in Figure 5-7, on the bottom of the dialog box we can see a Framework list box with the .NET Framework release versions available. Choose any one as a starting point and click OK.

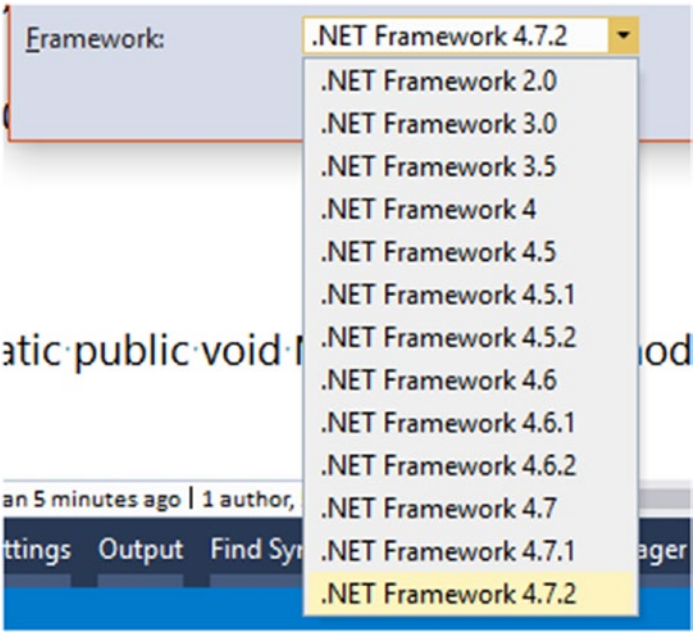


Figure 5-7. *List with .NET Framework versions available for.NET projects*

3. As shown in Figure 5-8, from the Solution Explorer window, open the `App.config` configuration file for the application.

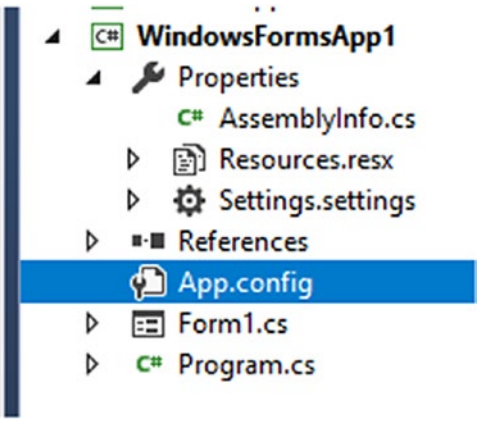


Figure 5-8. *The application configuration file created automatically by Microsoft Visual C#*

4. Listing 5-12 shows the content of `App.config` if we have chosen .NET Framework 4.*x*.

Listing 5-12. Configuration Values of App.config

```
<?xml version=”1.0” encoding=”utf-8” ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2" />
    </startup>
</configuration>
```

Understanding the <supportedRuntime/> Tag

The assembly of the project uses `version="v4.0"` for the CLR but with `sku=".NET Framework,Version=v4.7.2"` (the stock-keeping unit) as the specific value for the .NET Framework release version. That is, it includes the BCL (core set), the .NET Framework Class Library (complete set), the configuration, and the types in sync with the implementation of the tools and runtime. We can learn more information about `<supportedRuntime/>` in the official documentation at https://docs.microsoft.com/en-us/dotnet/framework/configure-apps/file-schema/startup/supportedruntime-element.

As shown in the Listing 5-13, we can clearly understand the relevance of the CLR versioning policy citing the `System.Text.StringBuilder` as example. If we use the `System.Text.StringBuilder` reference type, we must be aware that the implementation of the `System.Text.StringBuilder.Clear()` instance method has been available in the .NET Framework (complete set) since 4.0 version, but not in earlier versions. The `System.Text.StringBuilder.Clear()` instance method is just a convenience method, but the `System.Text.StringBuilder.Length` instance property has existed since the first implementation of the type and assigning the 0 (zero) value for the property has the same effect.

Listing 5-13. Excerpt of the Public Source Code for the System.Text.StringBuilder Reference Type

```
// Convenience method for [stringbuild instance].Length=0;
public StringBuilder Clear() {
    this.Length = 0;
    return this;
}
```

Because of such situations with common types or specialized types used by .NET software components and .NET software applications, the `<supportedRuntime/>` tag should be used by all applications since version 1.1 of the .NET Framework and should specify which is the release version of the CLR that the .NET software application supports. If it is possible, we should specify the SKU (stock-keeping unit) too. By default, if the `<supportedRuntime/>` tag is not present in the application configuration file, the versions of the CLR and .NET Framework used to build the application are used as the default values for the `<supportedRuntime/>` tag and their respective properties fields, `value` and `sku`. That is, the values for the tag and their properties (fields) are obtained by the VES from the assembly metadata.

In our experimental project, as we have chosen one of the version values in the Framework list box, based on the template of the project and in the rules of the Microsoft Visual Studio for project creation, not only is the CLR version chosen, but the most adequate versions of the assemblies are chosen automatically by Microsoft Visual Studio and included in the references of the project. Figure 5-9 shows the Reference Manager dialog box with the list of assemblies that currently are referenced by the project.

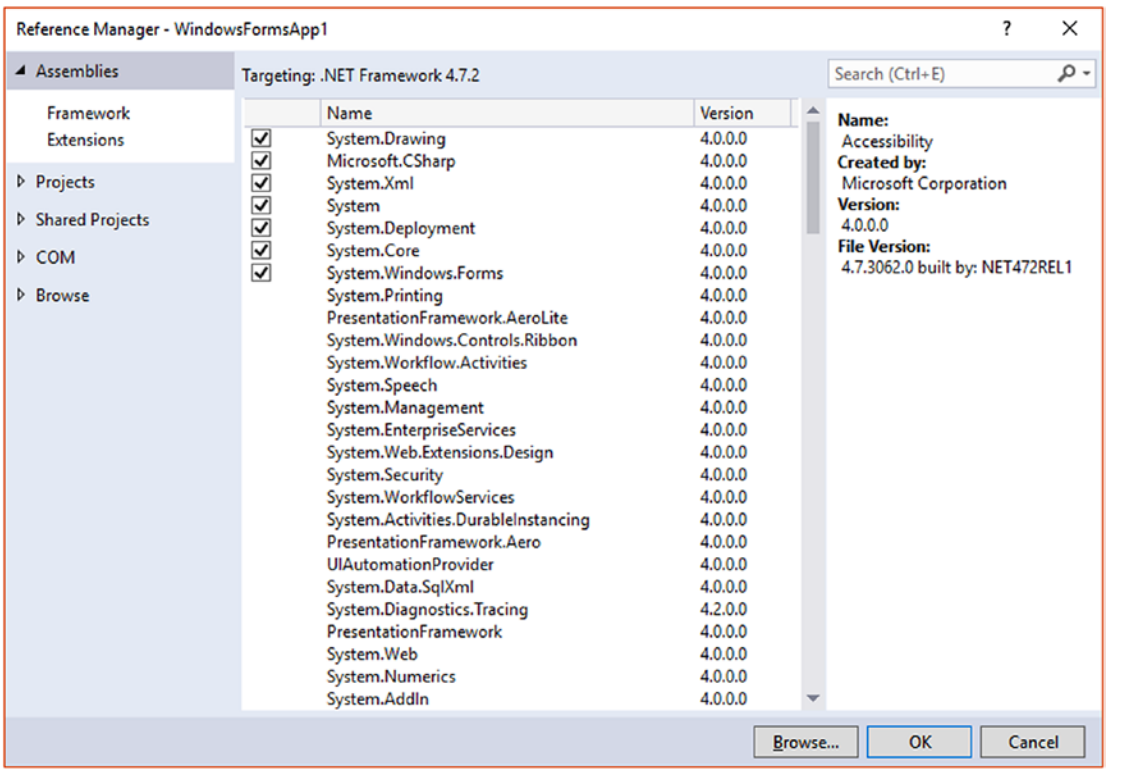


Figure 5-9. *List of assemblies referenced by the .NET project*

Now, with our experimental project compiled and the binaries created, we can use the ILDasm tool to see the assembly manifest of the assembly. As shown in Figure 5-10, we can see the `.ver` assembly directive with the 4.0.0.0 value `<major number>.<minor number>.<build number>.<revision number>` for all referenced assemblies. This is not only the version of the referenced assemblies but also the version number of the CLR that the referenced assemblies supports.

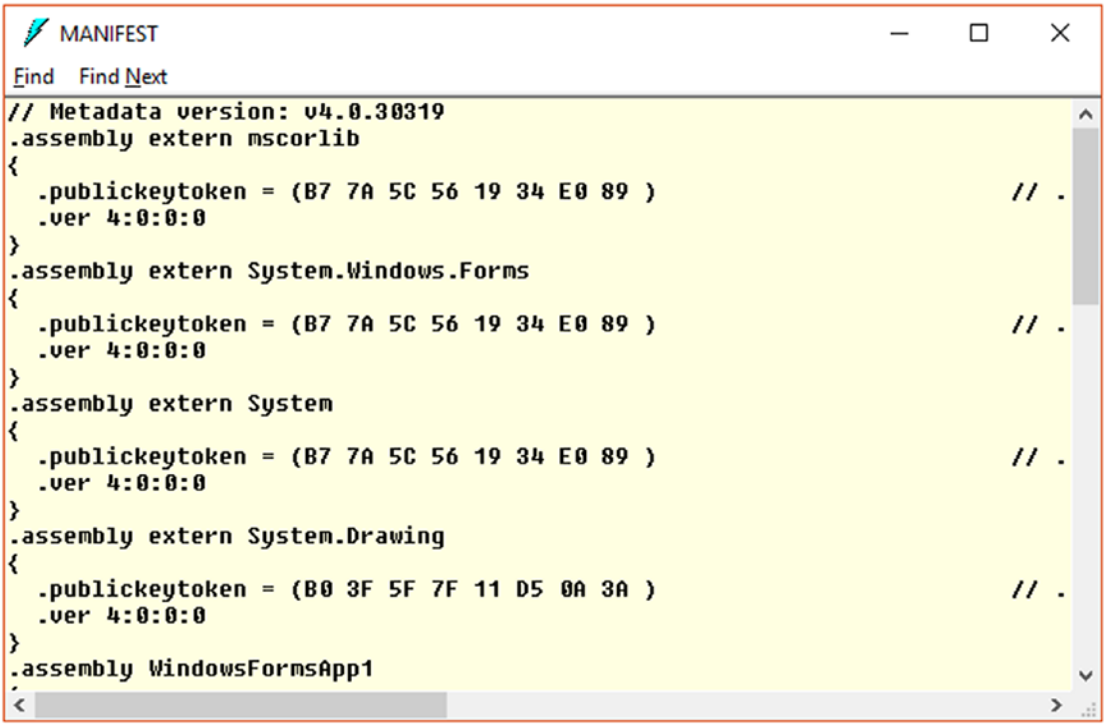


Figure 5-10. *The content of the assembly manifest and the .ver assembly directive for referenced assemblies*

Supporting Multiple Versions of the CLR

When multiple versions of the CLR should be supported by .NET components and .NET applications, we can have more than one `<supportedRuntime/>` tag in the configuration file. The first `<supportedRuntime/>` element should specify the priority runtime version, generally the version that application was built for, and the last `<supportedRuntime/>` tag must indicate the least compatible runtime implementation version. Using multiple `<supportedRuntime/>` tags is especially important with .NET Framework 1.1 through .NET Framework 3.5, because the BCL (core set) types have situations when members of certain types do not exist for a certain version but exist in .NET Framework 4.0 and newer. An example is the `System.Text.StringBuilder.Clear()` instance method, which exists in the implementation of BCL (core set) for .NET Framework 4.0 and newer.

With the .NET Framework 4.0 and newer, the `version` property value specifies the CLR release version, and the `sku` property value specifies a .NET Framework release version that the .NET software components and .NET software applications support.

Why is that?

By default, IDE-based products like Microsoft Visual C# and Microsoft Visual C++ use values in the assembly manifest and assembly metadata like the `.ver` assembly directive to set initial values in the configuration files. That is, if we chose any .NET Framework version 4.*, we are automatically choosing the CLR version 4.0.30319 (on my development machine it is the most recent version at that time), which we see in the application configuration file `<supportedRuntime/>` tag as `version="v4.0"`. For example, if we use .NET Framework 4.7.2 and CLR 4.0 for the design, implementation, and build of our project, even with the configuration `<supportedRuntime version="v2.0.50727"/>` tag in the application configuration file, the project compiles without errors. This is because most of development tools do not check these types of configuration aspects at compile/built time, mainly because one of the characteristics of the CLR design is to keep separate the influence of the configuration parameters used at execution time from the requirements of the metadata.

For example, using our experimental project compiled with .NET Framework 4.7.n and with support for CLR version 4.0, using the value `<supportedRuntime version="v2.0.50727"/>` tag, and running our experimental project from the environment of Microsoft Visual Studio, we will get an error similar to that shown in Figure 5-11. In the case of `System.BadImageFormatException`, this means the file format

of DLL (`.dll`) or EXE (`.exe`) does not conform to the format that the CLR expects. We can read details about this exception in the official documentation: https://docs.microsoft.com/en-us/dotnet/api/system.badimageformatexception?redirectedfrom=MSDN&view=netframework-4.7.2.

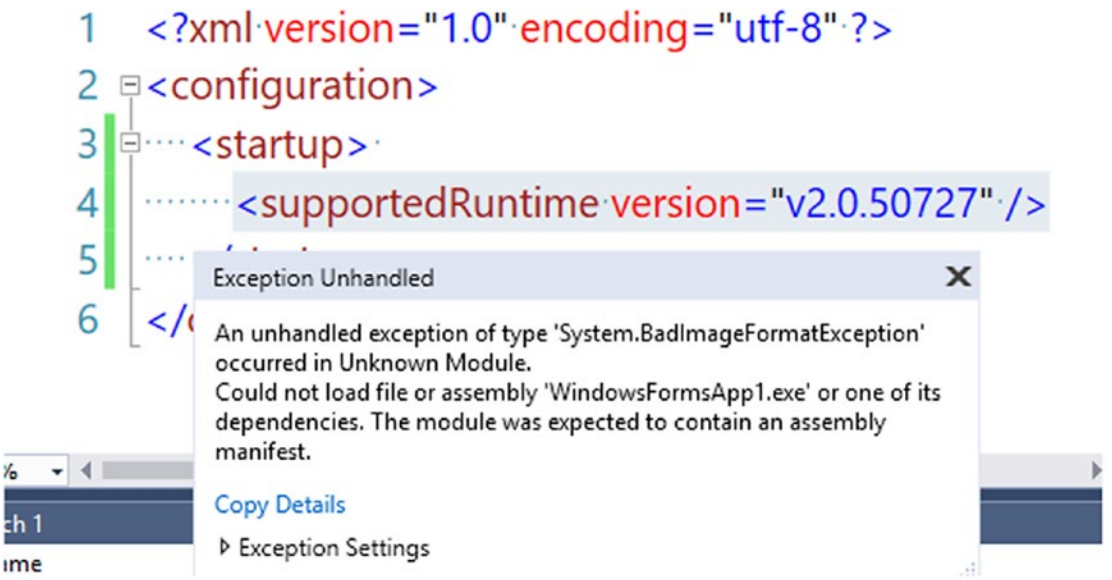


Figure 5-11. *Error by conflict generated by values in the application configuration file*

If we try to run the executable outside of the development environment, by default the managed process starts and finishes without any visible alert. But consulting the Microsoft Windows Event Viewer, we can see at least one entry for the application error, as shown in Figure 5-12.

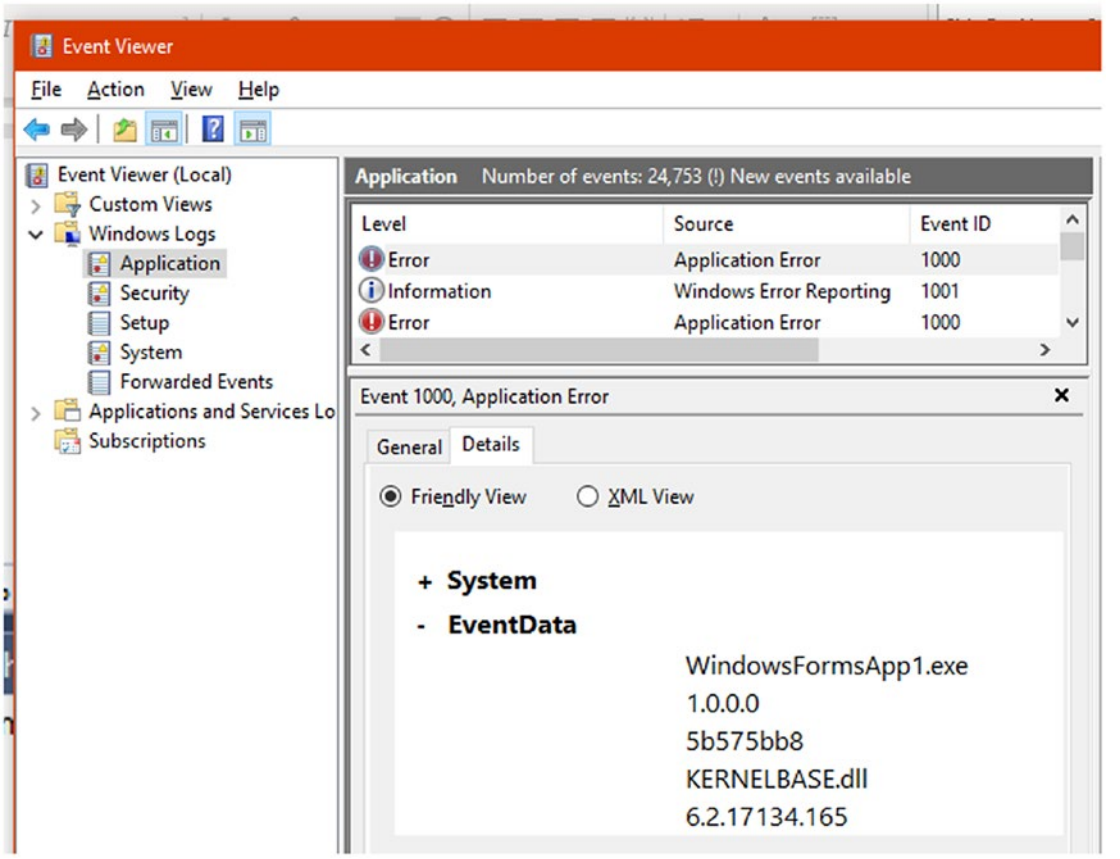


Figure 5-12. *Microsoft Windows Event viewer with an event error for the .NET application*

Working With CLR Version and SKU

Based on the rules of the CLI, the compilation tools should set this information in the assembly metadata. The `System.Runtime.Versioning.TargetFrameworkAttribute` custom attribute is applied automatically during the compilation and build process, independently of the values in the application configuration file for the `<supportedRuntime/>` tag.

Listing 5-14 shows how to get information from the assembly metadata. The `System.Reflection.Assembly` reference type uses the `Assembly.GetExecutingAssembly()` static method to get access to the instance of the assembly in execution. Using an instance of

the `System.Reflection.Assembly` reference type, we can use the `Assembly->ManifestModule` instance property to get a reference to the instance of the module (file) with the assembly manifest. The `Assembly->ManifestModule` instance property returns a reference to an instance of the `System.Reflection.Module` reference type.

Listing 5-14. Getting Information from the Assembly Metadata

```
/* Application configuration file without <supportedRuntime/> tag*/

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>

/* C++/CLI (Common Language Infrastructure) projection. */

Module^ _manifestModule{ Assembly.GetExecutingAssembly()->ManifestModule };

TargetFrameworkAttribute^ fx{ CustomAttributeExtensions.GetCustomAttribute
<TargetFrameworkAttribute^>( _manifestModule->Assembly ) };

/* C# programming language. */

/* Not using extension method. */
TargetFrameworkAttribute fx = CustomAttributeExtensions.GetCustomAttribute
<TargetFrameworkAttribute>( _manifestModule.Assembly );

/* Using extension method. */
//TargetFrameworkAttribute fx =  _manifestModule.GetCustomAttribute
<TargetFrameworkAttribute>();
```

As shown in Figure 5-13, running our experimental project and using the Microsoft Visual Studio Watch window with these two variables, `fx` and `_manifestModule`, we can see the values.

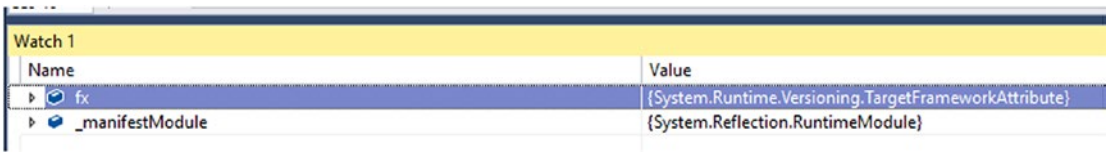


Figure 5-13. *Variables used to show information about assembly and the assembly manifest*

As shown in Figure 5-14 and Figure 5-15, by expanding the variable fx, we can see the `System.Runtime.Versioning.TargetFrameworkAttribute->FrameworkName` instance property with the value for the sku property of the `<supportedRuntime/>` tag. Expanding the variable `_manifestModule`, we can see the `System.Reflection.RuntimeAssembly->Assembly` instance property. Expanding the `System.Reflection.RuntimeAssembly->Assembly` instance property, we can see the `System.Reflection.RuntimeAssembly->ImageRuntimeVersion` instance property with the full version number of the CLR used to compile the assembly and module.

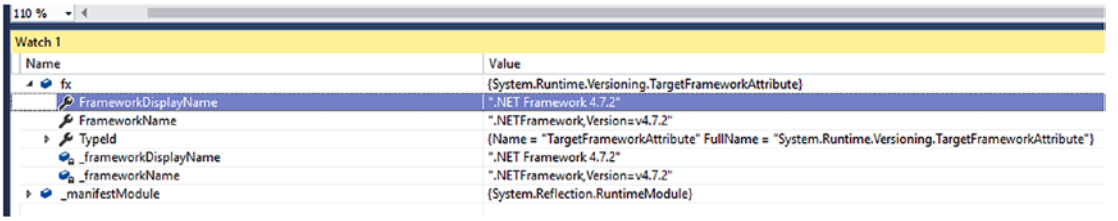


Figure 5-14. *Showing the information about the .NET Framework used to compile the application*

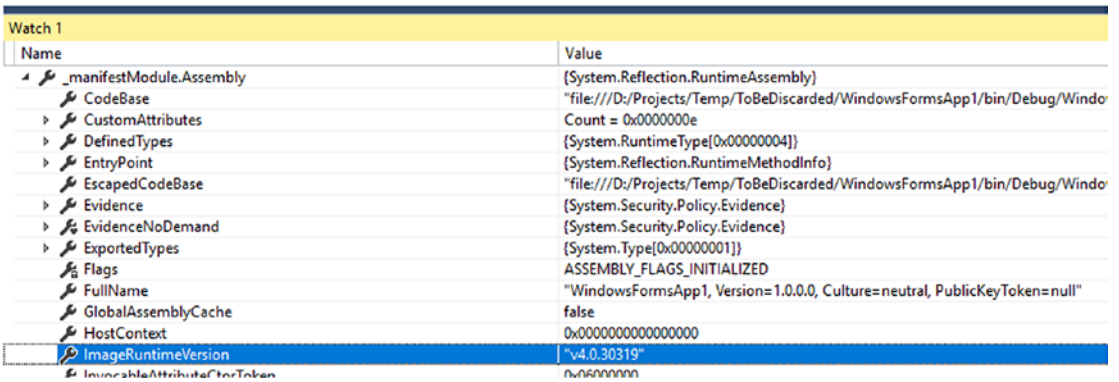


Figure 5-15. *Showing the information about the CLR version that the application was built for*

What we can remember is that by using these types of `System.Reflection`, we are reading the information from the assembly and module (file) of the .NET software components and .NET software applications, not from the application configuration file. This is one of the main responsibilities of the execution environment, that is, loading the information from the assembly metadata and module metadata with priority.

During the process of loading the modules and assemblies, steps like versioning binding are performed (we talk about this topic throughout the next chapter).

If we want to experiment with values for the `version` property and `sku` property of the `<supportedRuntime/>` tag, here are the values we should use to specify the adequate CLR release version depending on the .NET Framework release versions:

- With .NET Framework 1.0, use `version="v1.0.3705"`.
- With .NET Framework 1.1, use `version="v1.1.4322"`.
- With .NET Framework 2.0, use `version="v2.0.50727"`.
- With .NET Framework 3.0, use `version="v2.0.50727"`.
- With .NET Framework 3.5, use `version="v2.0.50727"`.
- With .NET Framework 4.0 through 4.7.2, use `version="v4.0"`.

Here we have a list with values for the `sku` attribute for .NET Framework 4.0 release version and newer:

- With .NET Framework 4.0, use `sku=".NETFramework,Version=v4.0"`.
- With .NET Framework 4.0 Client Profile, use `sku=".NETFramework,Version=v4.0,Profile=Client"`.
- With .NET Framework 4.0 Platform Update 1, use `sku=".NETFramework,Version=v4.0.1"`.
- With .NET Framework 4.0 Client Profile Update 1, use `sku=".NETFramework,Version=v4.0.1,Profile=Client"` Update.
- With .NET Framework 4.0 Platform Update 2, use `sku=".NETFramework,Version=v4.0.2"`.
- With .NET Framework 4.0 Client Profile Update 2, use `sku=".NETFramework,Version=v4.0.2,Profile=Client`.
- With .NET Framework 4.0 Platform Update 3, use `sku=".NETFramework,Version=v4.0.3"`.
- With .NET Framework 4.0 Client Profile Update 3, `sku=".NETFramework,Version=v4.0.3,Profile=Client`.
- With .NET Framework 4.5, use `sku=".NETFramework,Version=v4.5"`.

- With .NET Framework 4.5.1, use `sku=".NETFramework, Version=v4.5.1"`.
- With .NET Framework 4.5.2, use `sku=".NETFramework, Version=v4.5.2"`.
- With .NET Framework 4.6, use `sku=".NETFramework,Version=v4.6"`.
- With .NET Framework 4.6.1, use `sku=".NETFramework, Version=v4.6.1"`.
- With .NET Framework 4.6.2, use `sku=".NETFramework, Version=v4.6.2"`.
- With .NET Framework 4.7, use `sku=".NETFramework,Version=v4.7"`.
- With .NET Framework 4.7.1, use `sku=".NETFramework, Version=v4.7.1"`.
- With .NET Framework 4.7.2, use `sku=".NETFramework, Version=v4.7.2"`.

CHAPTER 6

Designing and Implementing Libraries

This chapter will cover some foundational elements of designing and implementing a .NET class library. We use .NET Standard 2.0 to create our .NET class library, and we will continue learning about aspects of versioning and version binding.

Metadata and Versioning

As we learned in Chapter 5, the metadata system and the VES coexist. Certain aspects of versioning are based on what is in the metadata in modules and assemblies, like we learned using the `.ver` assembly directive and the `System.Reflection.AssemblyVersionAttribute` and `System.Reflection.AssemblyFileVersionAttribute` custom attributes. In this chapter, we will learn more details about the behavior of the VES regarding the versioning system adopted by the platform.

The versioning system is not based only on the metadata of the modules and assemblies. The configuration files play a significant role in the various aspects of the versioning and management features within the platform.

Another two important aspects are that the CLR infrastructure consists of components mostly written in the C++ programming language and that the CLI specifies the foundational library called the BCL. This foundational library has several fundamental types. Some are built-in, and others are not. The `string`, `int32`, `char`, and `array` types are examples of built-in types specified by the CTS and implemented and supported directly within the VES (both part of the CLI). These built-in types have core

R. Villela, *Pro .NET Framework with the Base Class Library*, https://doi.org/10.1007/978-1-4842-4191-2_6

functionalities written in the C++ programming language and not exposed to the public, directly accessed, or directly manipulated by any managed programming language like the C# programming language or projections like C++/CLI. Despite this, managed programming languages such as the C# programming language and C++/CLI can access these built-in types and their functionalities to create .NET software components and .NET software applications using them.

This is possible because of the .NET libraries that exposes these internal functionalities written in the C++ programming language. Also, this .NET layer made of .NET libraries can aggregated functionalities to the fundamental data structures, data types, and behaviors internally written in a native programming language such as C++, or in a .NET programming language such as C#.

The main purpose of the built-in `string` data type and the `System.String` reference type of the BCL is to act as a container for the manipulation of sequences of values stored on instances of the `char` built-in data type, meaning the `System.Char` value type in the BCL. To achieve adequate results, the `System.String` reference type functionalities need to be coded in the C++ programming language using unsafe methods, other managed types of support, and built-in support from the VES. For example, the allocation of string buffers is made possible by the `String.FastAllocateString()` external static method, and the copy operations are created with the `String.wstrcpy()` static method, which is what we can call a specialized implementation of the Microsoft UCRT `wstrcpy()` function. In the `System.String` managed type, `String.FastAllocateString() method` is declared as internal and as external, and it is implemented in the C++ programming language as part of the built-in functionalities of the VES. In the case of the `System.wstrcpy()` method, the function is a wrapper-managed method, so it is declared and implemented as unsafe because it uses the `System.Buffer` managed type that specializes in doing operations using unmanaged pointers and operations such as copying and moving memory blocks allocated in the managed heap. A peculiarity of the `System.Buffer` managed type is that the methods have been designed to work only with arrays of primitive types, so internally all content of the array is manipulated using unmanaged pointers of the `System.Byte` (8-bit) value type. That is, the content of array is not treated as an instance of some managed type and just as a simple sequence of bytes.

Listing 6-1 shows the signature of the `System.String.FastAllocateString()` static method.

Listing 6-1. Declaration of the System.String.FastAllocateString() Static Method

```
[System.Security.SecurityCritical]  // auto-generated
[ResourceExposure(ResourceScope.None)]
[MethodImplAttribute(MethodImplOptions.InternalCall)]
internal extern static String FastAllocateString(int length);
```

The `String.FastAllocateString()` static method has `MethodImplAttribute` applied, and the constructor of the custom attribute receives the argument value `MethodImplOptions.InternalCall` of the `MethodImplOptions` enum type. The `MethodImplOptions.InternalCall` enum value means that the constructor and method with `MethodImplAttribute` applied are implemented as part of the CLR infrastructure subcomponents like the VES. Both types, the `MethodImplOptions` enum type and the `MethodImplAttribute` reference type, are part of the namespace `System.Runtime.CompilerServices` of the assembly `mscorlib` in file module `mscorlib.dll`.

Listing 6-2 shows the implementation of the `System.String.wstrcpy()` static unsafe method.

Listing 6-2. Implementation of the System.String.wstrcpy() Static Method

```
[System.Security.SecurityCritical]  // auto-generated
internal static unsafe void wstrcpy(char *dmem, char *smem, int charCount){

Buffer.Memcpy((byte*)dmem, (byte*)smem, charCount * 2); // 2 used
everywhere instead of sizeof(char)

}
```

As we can see in the listing, the method `String.wstrcpy()` does not call a public API of `System.Buffer` like the `System.Buffer.BlockCopy()` public method does; instead, it uses the `System.Buffer.Memcpy()` internal unsafe static method. This is possible because both types are declared on the same assembly with an internal access specifier, not necessarily in the same module like in this case. Remember, the internal accessibility level specifies that the type or member is visible and accessible to types within the same assembly. With the C# programming language, we can use the following combinations of keywords to declare a type or member with one of these internal accessibility levels:

- `internal` (assembly scope)
- `protected internal` (family or assembly scope)

Remember, within the C# programming language, the default accessibility level for a reference type and its members is private and for a value type and its members is public. However, we can explicitly declare the type and one or more of its members with an internal accessibility level, as shown in Listing 6-3, to conform to the implementation design. With the CIL code, for declaring the member of the type method with the internal accessibility level, we use the directive `.method` with the `assembly` attribute, as shown in Listing 6-4.

We can compile the CIL code in Listing 6-4 using these commands:

- `ilasm RVJ.Listing_6_3_A.il`
- `ilasm /X64 RVJ.Listing_6_3_A.il`

Listing 6-3. Method with assembly Accessibility Level (Example in the C# Programming Language)

```
namespace RVJ {

     public class A {

        protected internal void MethodOne(){}

    }

}
```

Listing 6-4. Method with assembly Accessibility Level: Managed Type A (Example in CIL)

```
.assembly extern mscorlib {
  .ver 4:0:0:0
}

.assembly RVJ.Listing_6_3 {
  .ver 1:0:0:0
}

.class public auto ansi beforefieldinit RVJ.Listing_6_3.A
       extends [mscorlib]System.Object {
```

```
.method famorassem hidebysig instance void
        MethodOne() cil managed {
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ret
} // end of method A::MethodOne

.method public hidebysig static void  Main() cil managed {
  .entrypoint
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ret
} // end of method A::Main

} // end of class RVJ.Listing_6_3.A
```

The member can be declared with accessibility level `protected internal` using the syntax and respective keywords via any high-level programming language with support for the CLR platform such as C#, F#, VB.NET, C++/CLI, and so on. Listing 6-5 shows the member, in this case an instance method, declared with the `protected internal` accessility level using the C# programming language syntax and respective keywords. Listing 6-6 shows how to apply the `protected internal` accessibility level in a member, also an instance method, using the CIL syntax and how to use `famorassem` (family or assembly) attribute.

Listing 6-5. Method with family or assembly Accessibility Levels: Reference Type A (Example in the C# Programming Language)

```
namespace RVJ {

       internal class A {

            protected internal void MethodOne() { }

      }

}
```

Listing 6-6. Method with family or assembly Accessibility Levels: Reference Type A (Example in CIL)

```
.assembly extern mscorlib {
  .ver 4:0:0:0
}

.assembly RVJ.Listing_6_5 {
  .ver 1:0:0:0
}

.class private auto ansi beforefieldinit RVJ.Listing_6_5.A
       extends [mscorlib]System.Object {

.method famorassem hidebysig instance void
        MethodOne() cil managed {
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ret
} // end of method A::MethodOne

.method public hidebysig static void  Main() cil managed {
  .entrypoint
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ret
} // end of method A::Main

} // end of class RVJ.Listing_6_5.A
```

However, when we use the C# programming language and the `internal` keyword to declare a top-level managed type, the things are a little different. As specified by CTS, formalized via the metadata, and put alive by the VES, there is no top-level managed type that can explicitly or implicitly be declared with an attribute assembly of `family OR assembly` or `family AND assembly`, like we do with the `.method` directive using the `assembly` and `famorassem` attributes, for example. A top-level managed type should be declared `public` or `private`. With the C# programming language, as shown in Listing 6-7, when a top-level managed type is declared with an access specifier keyword of `internal`, the compiler generates the CIL declaration of the managed type with the

private accessibility level, as shown in Listing 6-8. Currently, the C# programming language does not allow the use of the private keyword access specifier to declare top-level managed types, regardless of whether they're reference types or value types.

Listing 6-7. Using the internal Keyword of the C# Programming Language to Declare a Top-Level Managed Type

```
namespace RVJ {

      internal class A {

            protected internal void MethodOne(){};

      }
```

Listing 6-8. Using the private Attribute of the CIL to Declare a Top-Level Managed Type

```
.assembly extern mscorlib {
      .ver 4:0 : 0 : 0
}

.assembly RVJ.Listing_6_7 {
      .ver 1:0 : 0 : 0
}

.class private auto ansi beforefieldinit RVJ.Listing_6_7.A
extends[ mscorlib ]System.Object {

/* Method declared with famorassem attribute for the .method directive. */

.method famorassem hidebysig instance void MethodOne() cil managed {
      .maxstack  8
      IL_0000:  nop
      IL_0001 : ret
} // end of method A::MethodOne

.method public hidebysig static void  Main() cil managed {
      .entrypoint
      .maxstack  8
```

```
      IL_0000:  nop
      IL_0001 : ret
} // end of method A::Main

} // end of class RVJ.Listing_6_7.A
```

Metadata in the C++/CLI

With C++/CLI we can use the following keywords and combinations:

- `internal` (assembly)
- `protected public` (`famorassem`) (family or assembly)
- `public protected` (`famorassem`) (family or assembly)
- `protected private` (`famandassem`) (family and assembly)
- `private protected` (`famandassem`) (family and assembly)

When using access specifiers with two keywords, the most restrictive between both applies to the outside of the parent assembly, and the less restrictive between both applies to within the parent assembly. However, top-level managed types can be declared using only `private` or `public` keywords, meaning assembly or public accessibility levels. The examples in Listing 6-9, Listing 6-10, Listing 6-11, and Listing 6-12 show some of accessibility levels supported by C++/CLI and CIL.

Listing 6-9. Declaring a Top-Level Managed Type with an assembly (private) Accessibility Level and a Member Function with protected public (Example in C++/CLI)

```
namespace RVJ {
      namespace Listing_6_8 {
            private ref class A {

                  protected public:
                        void MethodOne() {};

            };
      };
};
```

Listing 6-10. Declaring a Top-Level Managed Type with an assembly (private) Accessibility Level and an Instance Method with the famorassem (family or assembly) Attribute (Example in CIL)

```
.assembly extern mscorlib {
      .ver 4:0 : 0 : 0
}

.assembly RVJ.Listing_6_9 {
      .ver 1:0 : 0 : 0
}

.class private auto ansi beforefieldinit RVJ.Listing_6_9.A
extends[ mscorlib ]System.Object {

.method famorassem hidebysig instance void  MethodOne() cil managed {
      .maxstack  8
      IL_0000:  nop
      IL_0001 : ret
} // end of method A::MethodOne

.method public hidebysig static void  Main() cil managed {
      .entrypoint
      .maxstack  8
      IL_0000:  nop
      IL_0001 : ret
} // end of method A::Main

} // end of class RVJ.Listing_6_9.A
```

Listing 6-11. Declaring a Top-Level Managed Type with an assembly (private) Accessibility Level and a Member Function with protected private (Example in C++/CLI)

```
namespace RVJ {
      namespace Listing_6_10 {
            private ref class A {
```

```
                protected private:
                      void MethodOne() {};

            };
      };
};
```

Listing 6-12. Declaring a Top-Level Managed Type with an assembly (private) Accessibility Level and an Instance Method with famandassem (family and assembly) Attribute (Example in CIL)

```
.assembly extern mscorlib {
      .ver 4:0 : 0 : 0
}

.assembly RVJ.Listing_6_11 {
      .ver 1:0 : 0 : 0
}

.class private auto ansi beforefieldinit RVJ.Listing_6_11.A
extends[ mscorlib ]System.Object {

      .method famandassem hidebysig instance void  MethodOne() cil managed {
      .maxstack  8
      IL_0000:  nop
      IL_0001 : ret
} // end of method A::MethodOne

.method public hidebysig static void  Main() cil managed {
      .entrypoint
            .maxstack  8
            IL_0000:  nop
            IL_0001 : ret
} // end of method A::Main

} // end of class RVJ.Listing_6_11.A
```

The Flexibility of the CLR

A managed environment is flexible and productive but has limits, as everything in life. But before achieving these so-called limits, we have tremendous possibilities for exploring design and implementation techniques. When designing and implementing .NET software components and .NET software applications, we should consider when to aggregate resources and when to extend the current .NET resource. To do this, we need to consider not only the C# programming language but the managed tools like C++/CLI and CIL, as well as native tools like the C++ programming language and the Assembly programming language.

An interesting example for this scenario is a typical operation with string buffers that are made by the `String.SubString()` instance managed method, as shown in Listing 6-13. The public method does the validations to guarantee that the argument values are correct and uses them to call an unsafe method with the name `String.InternalSubString()`.

Listing 6-13. Excerpt of Public Source Code for the System.String.SubString Instance Method

```
public String SubString(int startIndex, int length) {

if (startIndex < 0) {
               throw new ArgumentOutOfRangeException("startIndex", Environment.
               GetResourceString("ArgumentOutOfRange_StartIndex"));
            }

            if (startIndex > Length) {
                throw new ArgumentOutOfRangeException("startIndex",
                Environment.GetResourceString("ArgumentOutOfRange_
                StartIndexLargerThanLength"));
}

            if (length < 0) {
                throw new ArgumentOutOfRangeException("length",
                Environment.GetResourceString("ArgumentOutOfRange_
                NegativeLength"));
            }
```

```
        if (startIndex > Length - length) {
            throw new ArgumentOutOfRangeException("length", Environment.
            GetResourceString("ArgumentOutOfRange_IndexLength"));
        }
        Contract.EndContractBlock();

        if( length == 0) {
            return String.Empty;
        }

        if( startIndex == 0 && length == this.Length) {
            return this;
        }

        return InternalSubString(startIndex, length);
    }
```

An interesting aspect of `String.InternalSubString()` method is that it uses the two other unsafe methods, `String.FastAllocateString()` and `String.wstrcpy()`, as shown in Listing 6-14.

Listing 6-14. Excerpt of the Public Source Code for the Implementation of the String.InternalSubString Unsafe Method

```
 [System.Security.SecurityCritical]  // auto-generated
unsafe string InternalSubString(int startIndex, int length) {

        Contract.Assert( startIndex >= 0 && startIndex <= this.Length,
        "StartIndex is out of range!");
        Contract.Assert( length >= 0 && startIndex <= this.Length -
        length, "length is out of range!");

        String result = FastAllocateString(length);

        fixed(char* dest = &result.m_firstChar)
            fixed(char* src = &this.m_firstChar) {
                wstrcpy(dest, src + startIndex, length);
            }

        return result;
    }
```

.NET Standard 2.0

When we are designing a library, we are thinking of a piece of software that can be used for a reasonable commercial period, and we should assume a suitable number of target environments in the case of the .NET platform. Duplicating the code base is complicated and undesired, even with simple .NET class libraries.

Note The .NET Standard is a formal specification for a set of APIs implemented by the BCL of any implementation of.NET.

For example, here we are designing a library to support clients running at least .NET Framework 2.0; consequently, the CLR version is 2.0.50727. However, the intention of our library is to keep the clients on a migration path for .NET Framework 4.0 and CLR 4.0 as the minimum version. The objective is to explain the advantages of the .NET Framework 4 technologies. This will give our team a commercial advantage on the market. For example, one of the usual questions of development teams is how to work with the different .NET Framework 4.n versions. Considering that client development teams are new to the .NET Framework, this is a natural question. We can start explaining to the client development team a path for choosing the most adequate version of .NET Framework 4.n for the purposes of the project. We need to explain to the client development team, for example, that .NET Framework 4.0 has specialized versions for Microsoft Windows Client and Microsoft Windows Server.

Here is a list with the .NET Framework release versions and the values that we must use for the `version` attribute that specify the CLR release version:

- CLR release version
  - With .NET Framework 1.0, use `version="v1.0.3705"`.
  - With .NET Framework 1.1, use `version="v1.1.4322"`.

- With .NET Framework 2.0, use `version="v2.0.50727"`.
- With .NET Framework 3.0, use `version="v2.0.50727"`.
- With .NET Framework 3.5, use `version="v2.0.50727"`.
- With .NET Framework 4.0 through 4.7.2, use `version="v4.0"`.

Here is a list with values for the `sku` (stock-keeping unit) attribute for .NET Framework 4.0 release version and later:

- With .NET Framework 4.0, use `sku=".NETFramework,Version=v4.0"`.
- With .NET Framework 4.0 Client Profile, use `sku=".NETFramework, Version=v4.0,Profile=Client"`.
- With .NET Framework 4.0 Platform Update 1, use `sku=".NETFrame work,Version=v4.0.1"`.
- With .NET Framework 4.0 Client Profile update 1, use `sku=".NET Framework,Version=v4.0.1,Profile=Client"`.
- With .NET Framework 4.0 Platform Update 2, use `sku=".NETFrame work,Version=v4.0.2"`.
- With .NET Framework 4.0 Client Profile Update 2, use `sku=".NET Framework,Version=v4.0.2,Profile=Client"`.
- With .NET Framework 4.0 Platform Update 3, use `sku=".NETFrame work,Version=v4.0.3"`.
- With .NET Framework 4.0 Client Profile Update 3, use `sku=".NET Framework,Version=v4.0.3,Profile=Client"`.
- With .NET Framework 4.5, use `sku=".NETFramework,Version=v4.5"`.
- With .NET Framework 4.5.1, use `sku=".NETFramework,Version=v4.5.1"`.
- With .NET Framework 4.5.2, use `sku=".NETFramework,Version=v4.5.2"`.
- With .NET Framework 4.6, use `sku=".NETFramework,Version=v4.6"`.
- With .NET Framework 4.6.1, use `sku=".NETFramework,Version=v4.6.1"`.
- With .NET Framework 4.6.2, use `sku=".NETFramework,Version=v4.6.2"`.

- With .NET Framework 4.7, use `sku=".NETFramework,Version=v4.7"`.
- With .NET Framework 4.7.1, use `sku=".NETFramework,Version=v4.7.1"`.
- With .NET Framework 4.7.2, use `sku=".NETFramework,Version=v4.7.2"`.

The `version` and `sku` attributes are optional, but since the .NET Framework 4.0 simple values have been adopted, the use of the `sku` attribute is recommended. The first three RTM versions of the CLR use `<major>.<minor>.<build>` and `v1.0.3705`, `v1.1.4322`, and `v2.0.50727` for the `version` attribute value. Since .NET Framework 4.0 only the `<major>.<minor>` version numbers are required; that is, use `v4.0` instead `v4.0.30319` for the `version` attribute value. One important thing to remember is that the `version="v2.0.50727"` attribute of the CLR is used for .NET Framework 3.0 and 3.5 too.

The Relevance of a Versioning Policy

We must explain to the clients of our .NET libraries (even if they are part of the same corporation) that not only the values of the assembly manifest metadata will represent the version of the .NET software components and .NET software applications that we design and implement, but the configuration files are fundamental pieces of the .NET Framework way of working, that is, architecture and engineering. In other words, the CLR recognizes the existence or not of the application configuration file and other configuration files, working through components specially designed to interact with these XML-based files. For example, the values within the application configuration file that override the .NET version information presented earlier are checked by the execution environment to make certain decisions. Other aspects about configuration such as the locations (folders) of configuration files are specified by rules, and the name scheme also is specified by rules. That is, these elements are *not* based on the desire of someone in the company. If we create as output a .NET binary executable (EXE), the configuration should be present in the same folder (or subfolder) as the binary executable and with the same name as the binary executable, but the last part of the name (known as the extension) should be the string `.config`. For example, for a binary executable with the output name `SupportIssue.Exe`, the configuration file name will be `SupportIssue.Exe.config`; this file is called an *application configuration file*. If the output .NET binary executable is a dynamic link library (DLL), the application

configuration file is automatically assumed by the code within the DLL. The execution environment also checks the information obtained from the machine configuration file. This machine configuration file has a standard name, `machine.config`. The machine configuration file exists on the local machine (computer) in the subfolder `Config` of the root folder where the runtime is installed, as shown in Figure 6-1, Figure 6-2, and Figure 6-3. Another important aspect of the behavior of the execution environment is that the settings in `machine.config` apply to all environments on the local computer. For example, if we have the same settings for the binding to a specific assembly version in a machine configuration file and in an application configuration file, the settings in the machine configuration file take precedence over all the other configuration settings. We should be aware of these behaviors of the CLR so we can create virtual machines to simulate configuration scenarios, prepare our libraries of components with predictable actions, and document these predictable actions at various levels as appropriate (development, infrastructure management, and application user).

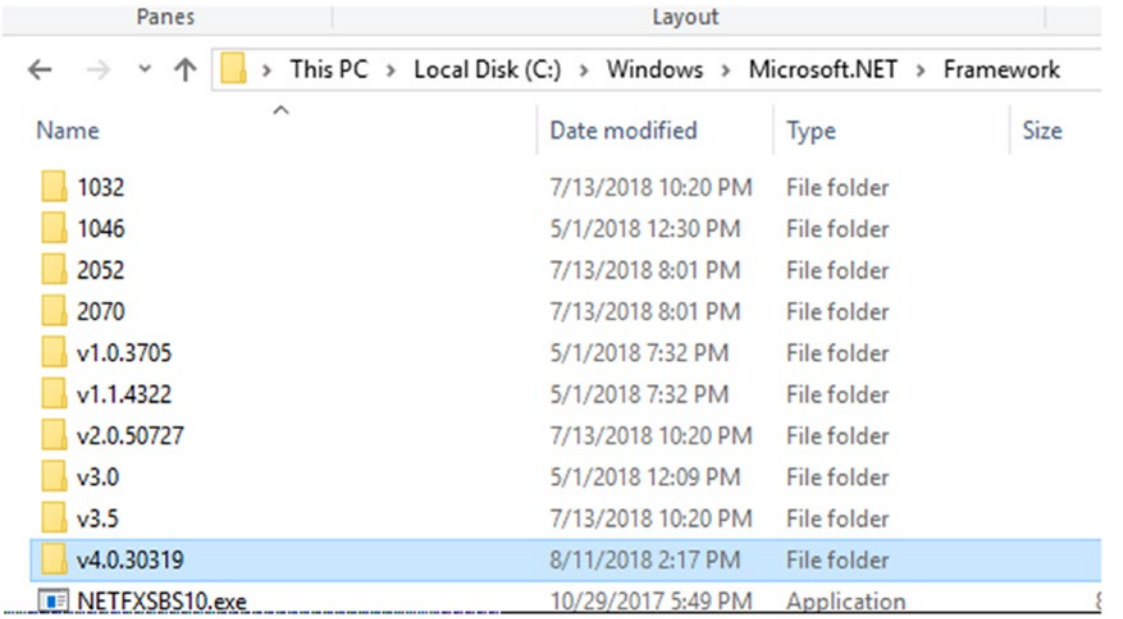


Figure 6-1. *Installation folder of .NET runtimes*

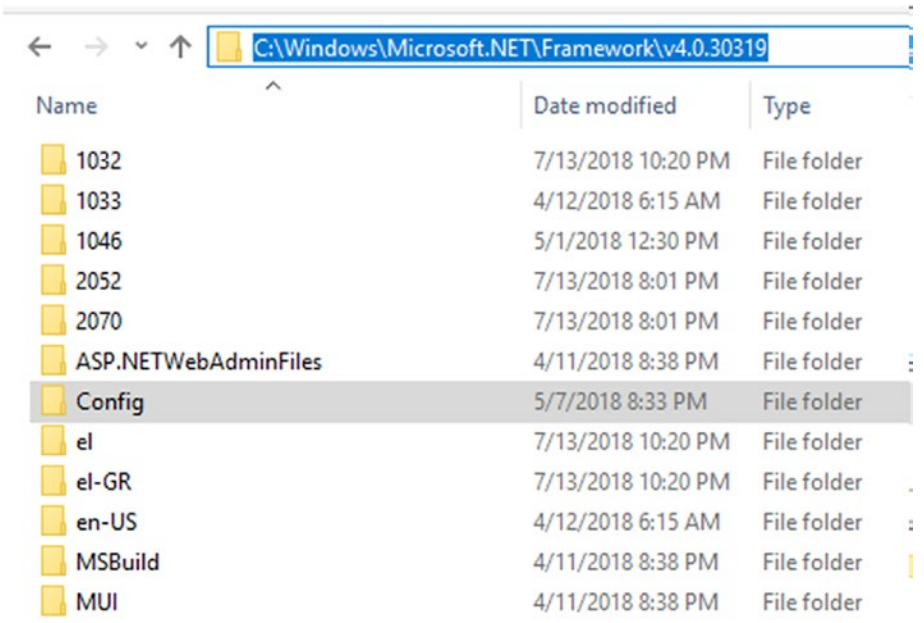


Figure 6-2. *.NET runtime Config file folder*

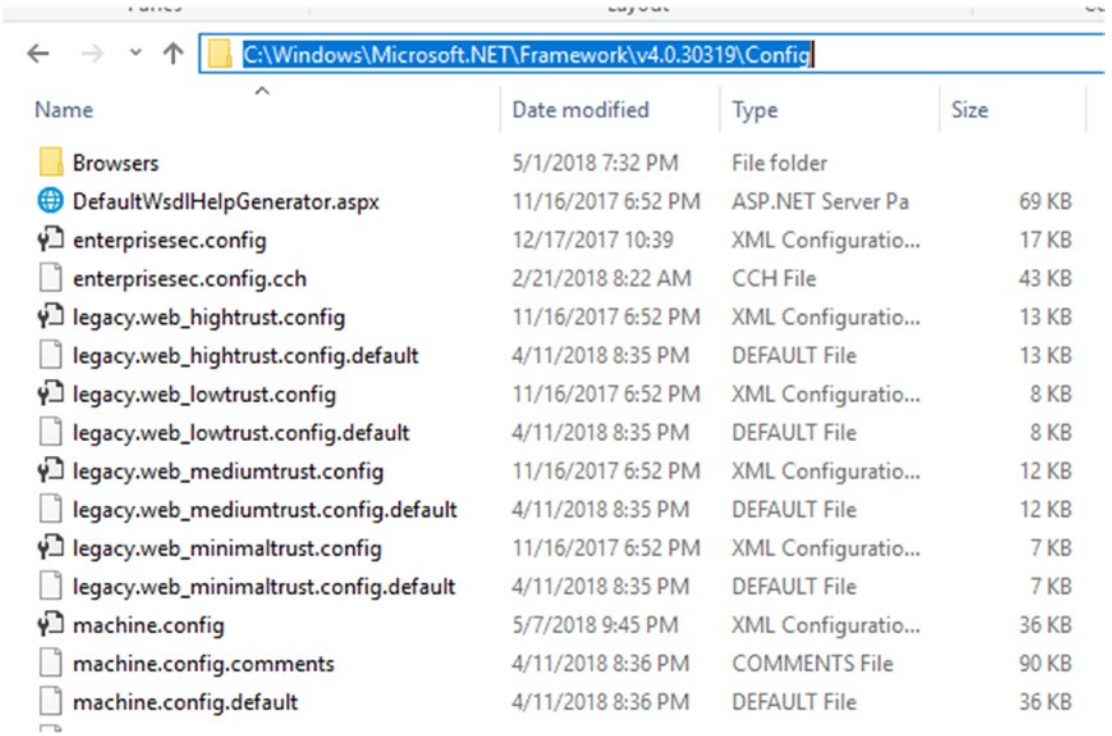


Figure 6-3. *Configuration files in .NET Runtime config folder*

We can open these config files using a code editor, like the ones that are part of Visual Studio or Microsoft Visual Code. When doing any basic .NET experiment, we will learn that the XML tags for configuration are in fact managed types, most of them reference types.

So, it is fundamental to understand that for every piece in the configuration file, we will have instances of managed types, reference types, and value types in general. Also, having complex content for configuration files is not useful; "strange" combinations just to prove that we could do something better. Do not create erroneous expectations that we will do all things correctly on the first attempt. This is a common mistake made over and over by environments and people. When working with powerful technological environments like Windows and the .NET Framework, we cannot make decisions in an isolated manner. Versioning, for example, requires a group of related resources and contexts. One of the great lessons that we should accept when starting to work with Microsoft Windows and the .NET Framework is that knowledge is not an individual "desired way" of doing things; we need real knowledge about the platforms.

Creating a .NET Standard Library

Now let's start a tutorial to create a .NET Standard library. We are using Visual 2017 15.8 or newer.

Using File ➤ New Project ➤ Visual C#, choose .NET Standard; the New Project dialog opens, as shown in Figure 6-4.

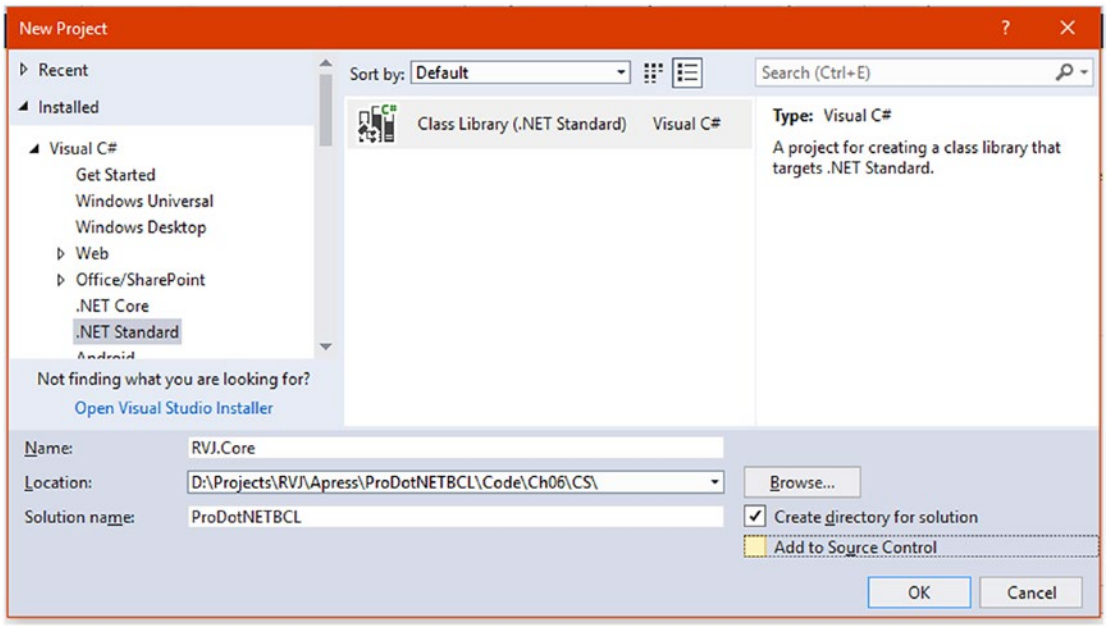


Figure 6-4. *New Project dialog for class library using the .NET Standard*

Choosing the .NET Standard Version

By default, Visual Studio 2017 chooses the most recent version installed on the development machine; this is the current model. We can change the .NET Standard version using the Project Properties window after the project created. For our project, we are using the value **Class Library** for the output type and the value **.NET Standard 2.0** for the target framework. In the Solution Explorer window, we can see the .NET Standard metapackage has the name `NETStandard.Library`.

After the project is created, open the Solution Explorer window, choose the project RVJ.Core, and open the project properties. Figure 6-5 shows the project properties on the Application tab, and we can set the properties of the project.

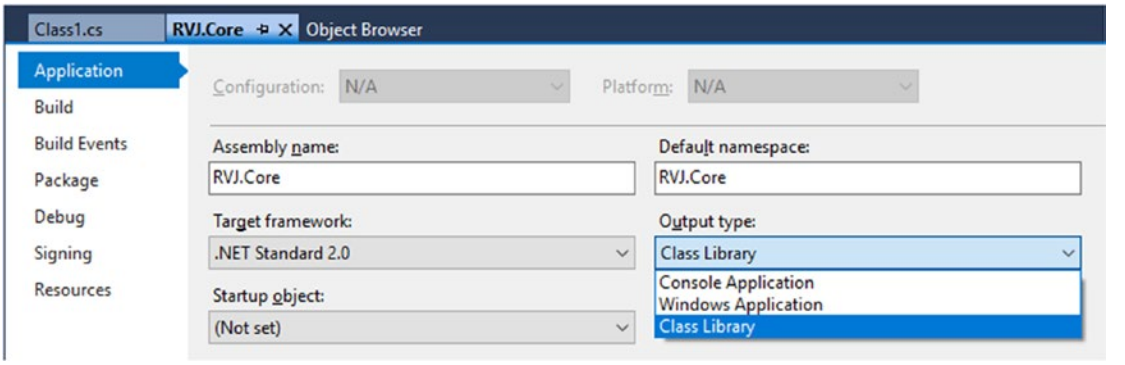


Figure 6-5. *Project properties for the .NET Standard*

Next, on the Application tab of the project properties, as shown in Figure 6-6, choose one of the values, like .NET Standard 1.0, in the list box "Target framework" and see what happens.

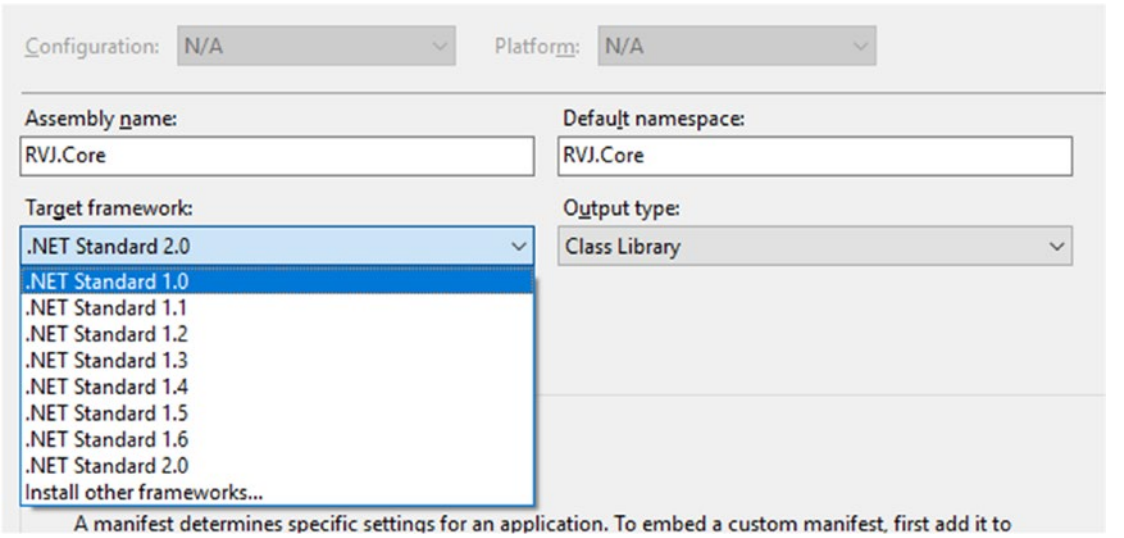


Figure 6-6. *Project properties with the list of .NET Standard versions*

Now we can include the code for our library and build the library.

Metapackages

A *metapackage* is a new conceptual model to organize the packages that make up the libraries and their components (assemblies, modules, and resources) in a more manageable way. As shown in Figure 6-7, the SDK for .NET Standard 2.*x* consists of the metapackage `Microsoft.NETCore.Platforms` and the assembly `netstandard.dll`. Using the Application section in the Project Properties window, if we change the value of the "Target framework" field, after a few seconds Visual Studio changes the items in the dependencies list.

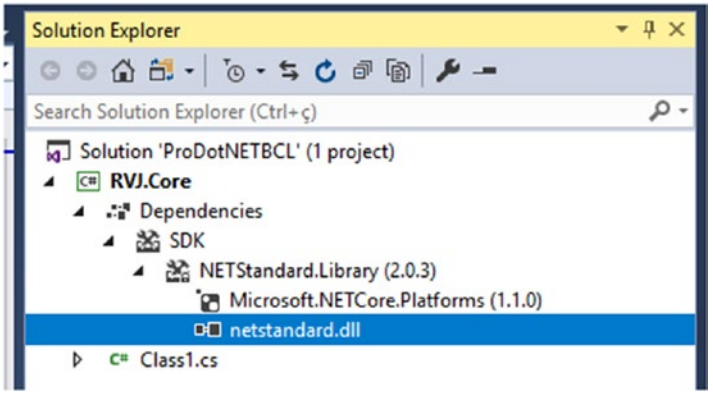


Figure 6-7. *List of metapackages and packages*

As shown in Figure 6-8, by expanding the dependencies list, we can see the metapackage `NETStandard.Library`, all the packages, and the relationships between these packages.

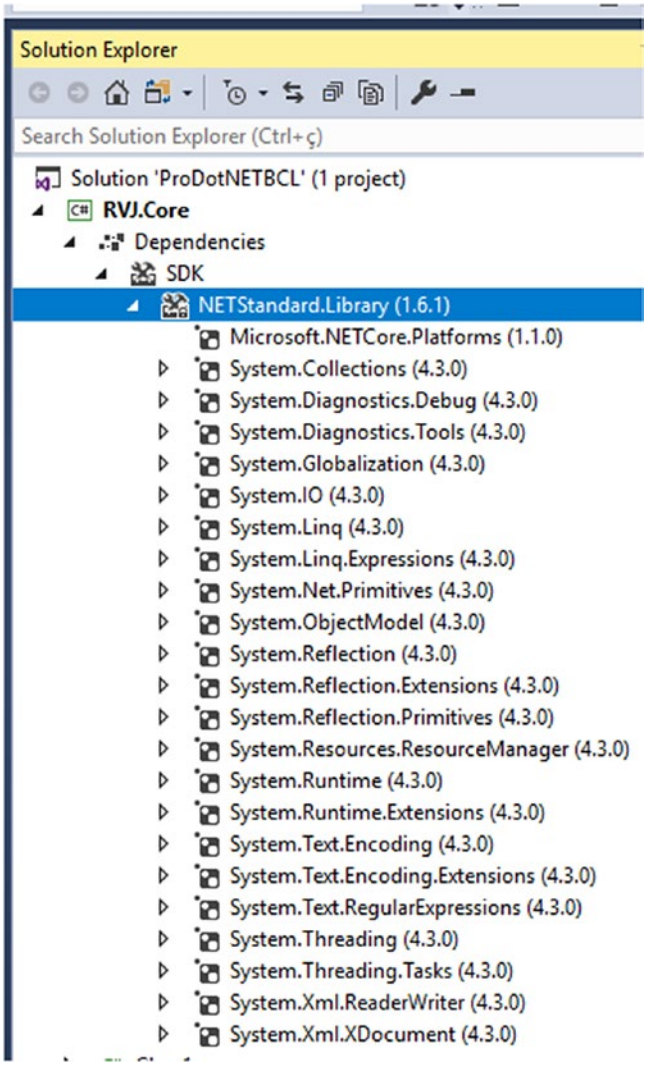


Figure 6-8. *List of content of packages*

As shown in Figure 6-9, the `System.Collections` package references the packages `Microsoft.NETCore.Platforms`, `Microsoft.NETCore.Targets`, and `System.Runtime`; it also has the assembly `System.Collections.dll`. We can change the versions of the .NET Standard to see the packages in each version.

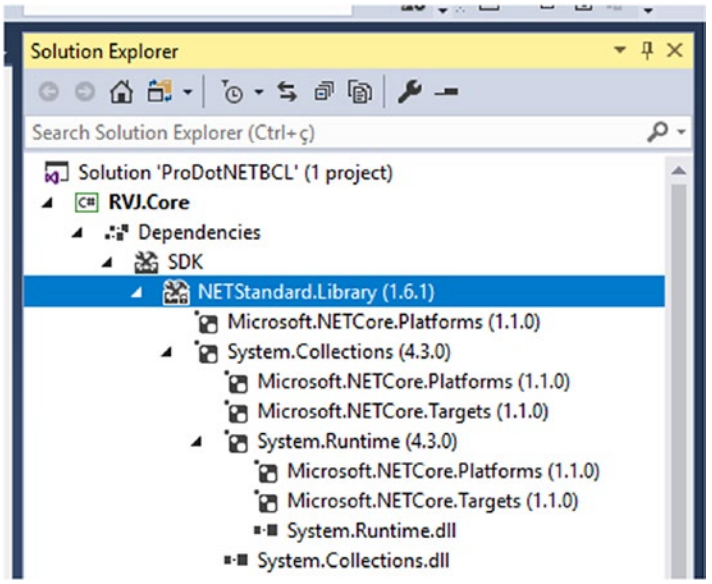


Figure 6-9. *List of content of packages*

Examining Packages

The packages are NuGet packages; the means they follow the same model adopted by .NET Core, which is part of the foundation of the .NET Standard.

One of the fundamental ideas of the package concept is to clearly organize the support for one or more .NET traditional framework implementations and the new package-based framework implementations.

Traditional implementations includes .NET Framework 4.0, .NET Framework 4.5, .NET Framework 4.6, .NET Framework 4.7, and so on. The package-based framework is a new model for the design, implementation, and deployment of the frameworks. Until the .NET Standard, we do not have any form to describe these relations using some form of metadata. For example, if we are designing a set of assemblies, how do we package these assemblies and describe their relations? Some of these assemblies were made specifically for use with the full implementation of the .NET platform using Microsoft Windows–specific technologies, meaning the .NET Framework (complete set). Other sets of these assemblies were made to work with .NET Core on mobile platforms and across platform (via Xamarin, for example).

The Assembly Version and Assembly File Version

Now, we will change back to the .NET Standard 2.0 and, with the class library compiled, use the ILDasm tool to load the assembly manifest. Again, the `.ver` assembly directive is used to set the assembly version. As shown in Figure 6-10, we can set the values for the `.ver` assembly directive using the Project Properties window; choose the tab Package and set the values of the four fields at the bottom of screen.

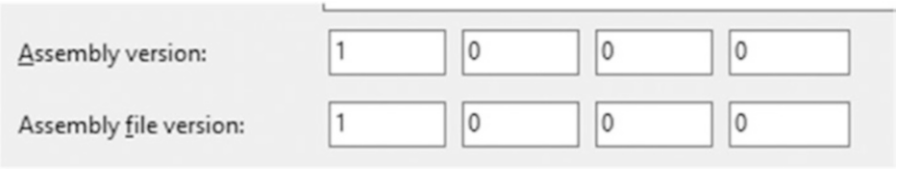


Figure 6-10. *Project properties with assembly version and assembly file version*

The default rules implemented in the VES are not the only aspect that helps us make decisions about versioning. The CLR configuration files are also part of that equation. The configuration files in the .NET Framework are classified by context. For example, there are machine configuration files, application configuration files, and security configuration files, and all use the XML format to describe the elements (tags), their properties (fields), and their values (instances of data types), and being in an XML format, the structural and behavioral rules are described by one or more XML schema definition (XSD) files. The configuration files are manipulated using specialized BCL types available in the `System.Configuration` assembly (`System.dll`, `System.Configuration.dll`, and `System.Configuration.ConfigurationManager.dll`). In fact, it's more correct to say that the BCL configuration types *may* have a representation in an XML format because not all BCL configuration types have one. The `System.Configuration` assembly has fundamental BCL configuration types, and specialized technological environments such as (but not only) ASP.NET, .NET Core, ASP.NET Core, Xamarin.Mac, Windows Forms, WPF, WCF, WF, WIF, and UWP, and Microsoft Azure technologies create their own configuration types and respective elements (tags) based on the infrastructure of BCL configuration types. For example, the `System.Configuration.ApplicationSettingsBase` abstract reference type exists to serve as the base class for derived classes implementing a feature called *application settings* that is used by Windows Forms. Listing 6-15 shows the `System.Configuration.ApplicationSettingsBase` abstract reference type.

The System.Configuration assembly is available in the .NET Framework, ASP.NET, .NET Core, ASP.NET Core, and Xamarin.Mac.

Listing 6-15. Declaration of the System.Configuration.ApplicationSettingsBase Abstract Reference Type

```
/* C++/CLI (Common Language Infrastructure) projection. */

public ref class ApplicationSettingsBase abstract : System.Configuration.
SettingsBase, System.ComponentModel.INotifyPropertyChanged

/* C# programming language. */

public abstract class ApplicationSettingsBase : System.Configuration.
SettingsBase, System.ComponentModel.INotifyPropertyChanged
```

System.Configuration.ApplicationSettingsBase derives from the base class System.Configuration.SettingsBase abstract reference type and is used to support the design and implementation of specialized mechanisms to manage user property settings. Two examples of specialized mechanisms that manage user property settings are ASP.NET Web Forms and Windows Forms; both use these types as the settings infrastructure of their customized implementation mechanisms to manage user settings.

Listing 6-16 shows an example of a typical application configuration file with for the .NET Framework and the CLR.

Listing 6-16. Startup Settings Configuration File

```
<?xml version=”1.0”?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2"/>
  </startup>
</configuration>

<!-- With more than one <supportedRuntime/> element -->

<?xml version=”1.0”?>
<configuration>
  <startup>
```

```
    <!--
<supportedRuntime  version="v4.0" sku=".NETFramework,Version=v4.7.2"/>
    <supportedRuntime  version="v4.0" sku=".NETFramework,Version=v4.7.1"/>
    <supportedRuntime  version="v4.0" sku=".NETFramework,Version=v4.7.1"/>
-->

    <supportedRuntime  version="v2.0" sku=".NETFramework,Version=v2.0"/>
<!--
<supportedRuntime  version="v2.0.50727" sku=".
NETFramework,Version=v4.7.2"/>
-->
<!--
<supportedRuntime  version="v1.1.4322" sku=".NETFramework,Version=v1.1.4322"/>
-->
<!--<supportedRuntime  version="v1.0.3705" sku=".NETFramework,Version
=v1.0.3705"/>
-->
  </startup>
</configuration>
```

Working with Version Numbers

During the lifetime of a library, we should make decisions about when we change the version number. The version number of an assembly is organized into four numbers of `System.Int32`, as shown in Figure 6-11. So, the range of values is enough for most .NET software components and .NET software applications. According to the CLI, the version number should be captured at compile time and used as part of all references to the assembly within the compiled module.

The number is structured like `major.minor.build.revision`.

- Major version number
  - The first of these four digits is considered a major version number.
  - When we need to rewrite many fundamental aspects of the API of the libraries and cannot guarantee backward compatibility, an update of the major version number should be considered.

    - Assemblies with the same name but with different major versions are not interchangeable.
- Minor version number
    - The second of these four digits is considered a minor version number.
    - Assemblies with the same name and same major number but with different minor versions should be used when significant enhancements (not bug fixes) are introduced.
    - This kind of version number change should be used when the goal is backward compatibility.
- Build version number
    - The third of these four digits is considered a build version number.
    - The assemblies changing should be represented only by a recompilation of the same source code.
    - Including support for a compiler tool, a software platform, and/or a processor should be considered scenarios for the update of this number.
- Revision version number
    - The fourth of these four digits is considered a revision version number.
    - Assemblies with the same name, same major version number, and same minor version number but with different revisions are characterized as being fully interchangeable.
    - The typical scenario for this is to fix a bug.

We should be aware that the module version number, assembly version number, and now the package version number do not need to have the same sequence for any of parts, as shown in Figure 6-11 and Figure 6-12. In fact, the package version number uses only three parts and not four parts, but we should specify standard project rules for version number parts so as to not confuse the dependencies. For example, it is adequate

to keep the version number of the package not based on the version on the assembly/module version number. The package version number should be based on rules of the build process plus a set of project product rules, like including or removing features and including or removing assemblies.

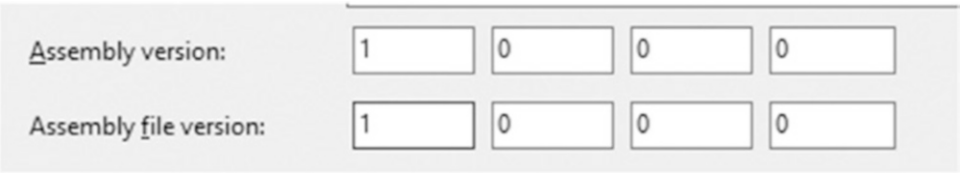


Figure 6-11. *Assembly version number/module version number*

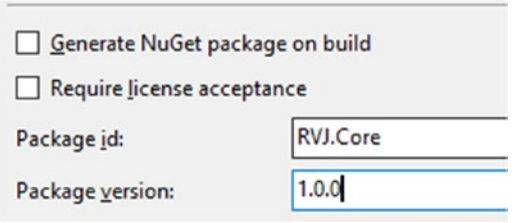


Figure 6-12. *Package version number*

Project Properties

After the implementation of the code for the library, we can generate a NuGet package. But before the generation of the NuGet package, and in fact during the development cycle of the .NET software component, we should define certain characteristics, and some of them should be based on one or more Microsoft Visual Studio Project templates. For example, when we open the project properties in Microsoft Visual Studio, certain values appear on the following tabs: Application, Build, Build Events, Package, Debug, Signing, and Resources.

As shown in Figure 6-13, on the Application tab, we can define the assembly name, default namespace, target framework, and output type. At a certain point, these should be blocked for update via the Microsoft Visual Studio IDE. This can be updated only via the build process using specialized scripts and tools.

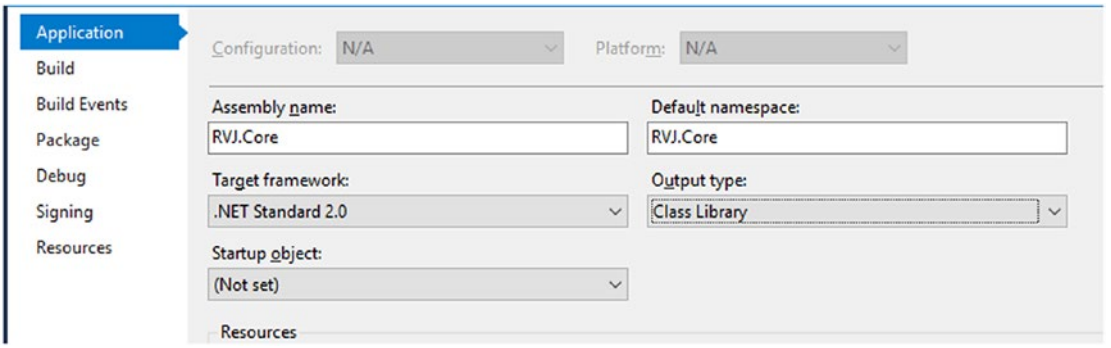


Figure 6-13. *Project Properties / dialog, Application tab*

In the Build tab, as shown in Figure 6-14, we can set the values for the typical options of the configuration of the project, such as the support for unsafe code, and clicking on the **Advanced...** button we can even alter the C# programming language version that should be used in the project, as showed in the Figure 6-15. This decision could be valid for all components in the library or supported on a case-by-case basis, like including support for certain code file implementations. Certainly these decisions will be made and assumed via the build process, via scripts, and/or via specialized tools.

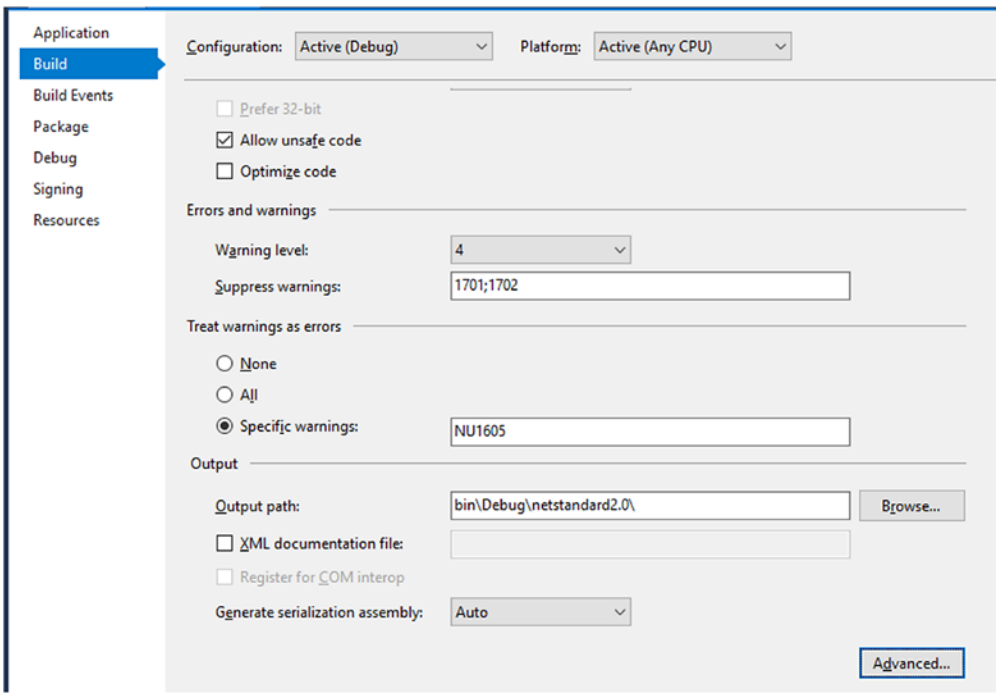


Figure 6-14. *Project configuration, Build tab*

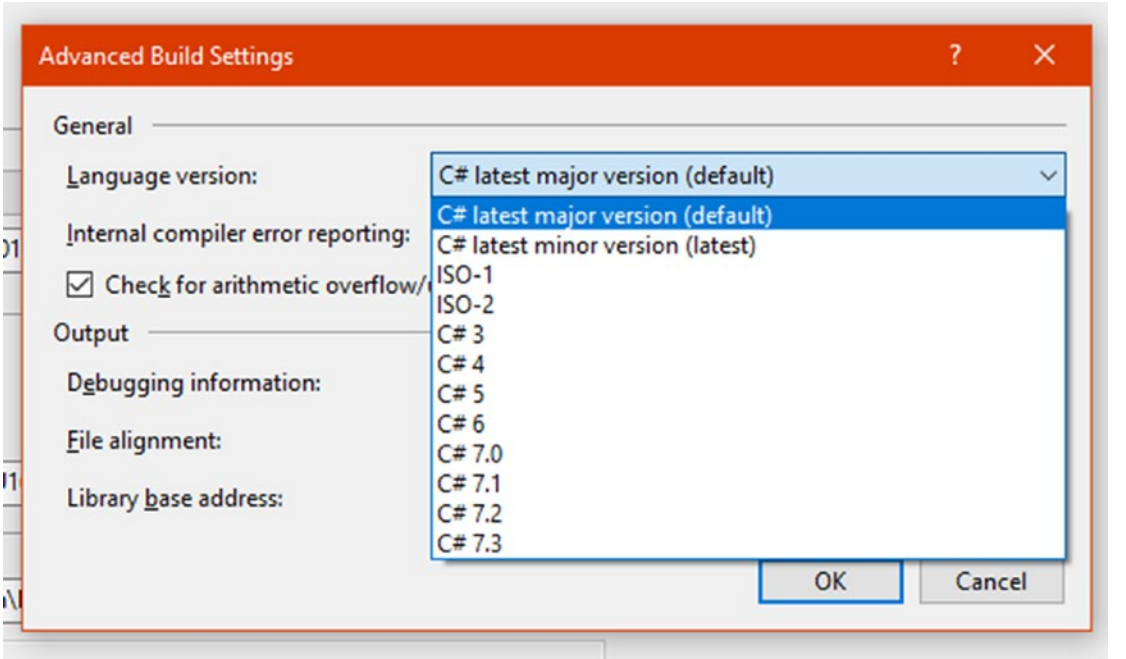


Figure 6-15. *Build tab, advanced configuration (via the Advanced button)*

These two tabs and some properties are a few examples of the elements that we should define and certainly will be updating during the lifecycle of the .NET software component and .NET software application.

Packaging

After developing the .NET software component and .NET software application and testing the packaging, we can use the menu Build ➤ Pack [Package Name], as shown in Figure 6-16.

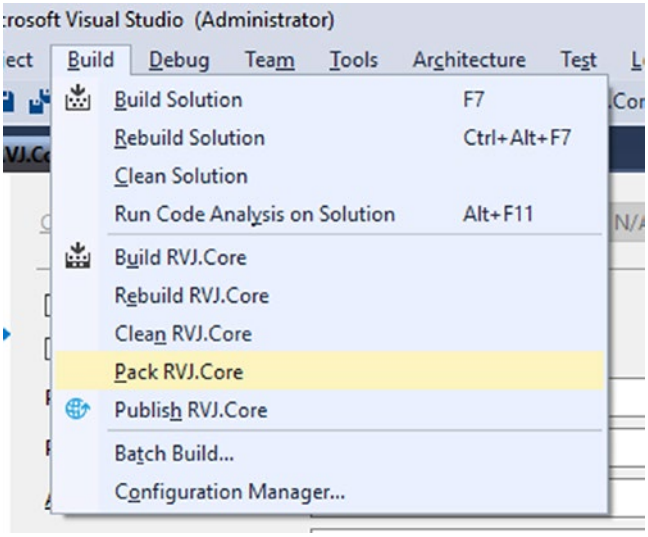


Figure 6-16. *Pack option of the Build menu*

As shown in Figure 6-17, when the project is compiled without any problems, we can check the output binary folder to find the file with the extension `.nupkg`. If we alter the extension to be `.zip`, the Microsoft Windows File Explorer changes the icon and associates the file with software that can manage files in this format. Looking at the folder structure inside the NuGet package file, we can see a sequence of folders: `lib`, then `netstandard2.0`, and then `RVJ.Core.dll`. That way we can more easily locate the assemblies/modules and the .NET Standard used to build the package.

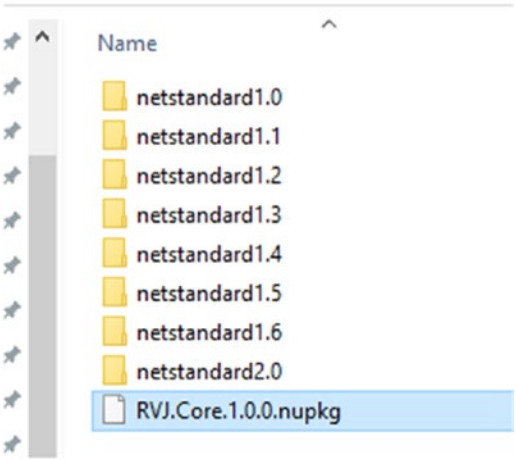


Figure 6-17. *NuGet package generated from the project*

To deploy a .NET software component based on the NuGet standard, we need at least three repositories: a local development/debugging repository, a development test repository (local or not), and a production repository. If we want to debug our project on our development workstation, we can use a local folder as a local repository. For example, as shown in Figure 6-18, considering a sample Windows Forms project, click the project properties and choose Manage NuGet Packages, as shown in Figure 6-19.

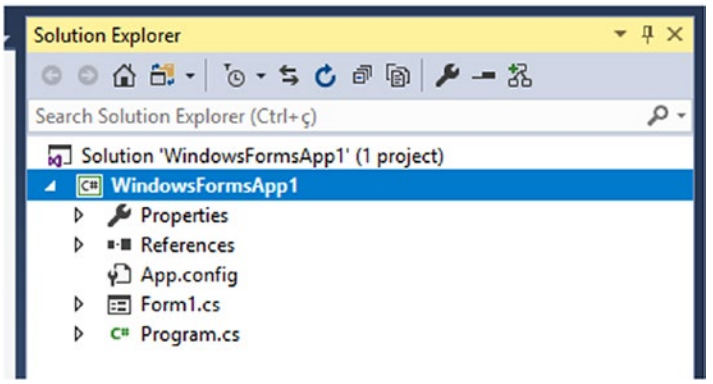


Figure 6-18. *Sample application without NuGet package included*

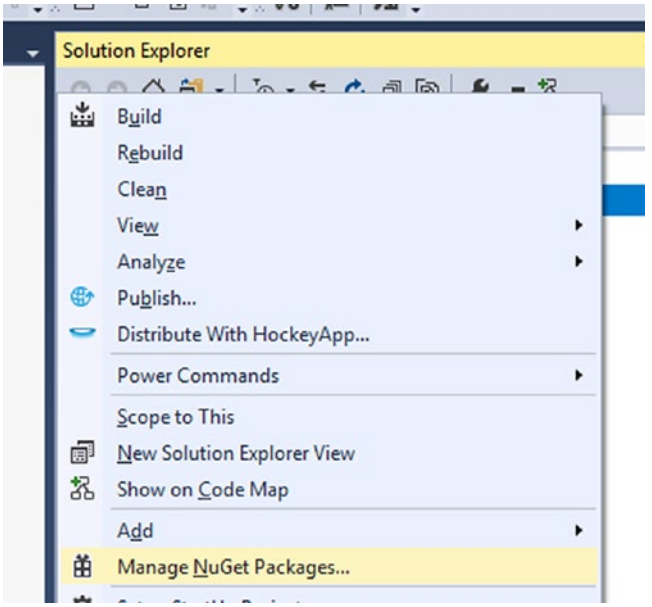


Figure 6-19. *Manage NuGet Packages on the Build menu*

As shown in Figure 6-20 at the right top, we can see the list box "Package source" and the gear icon (Settings) on the right side. Click the gear icon. Here we can insert, update, and remove sources where the NuGet packages are stored and accessible. After created a source called ProDotNetBCL indicating where our packages are stored, we can refresh the sources, and our package appears. As we can see in the package properties, the version information is available.

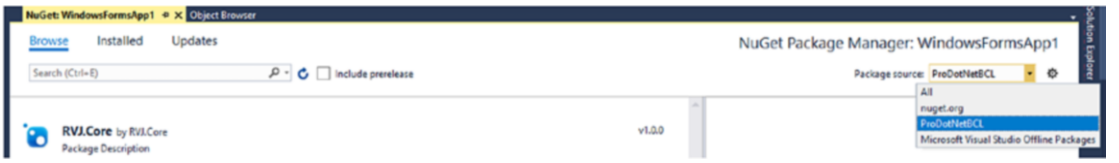


Figure 6-20. *NuGet Package Manager window in Microsoft Visual Studio*

As shown in Figure 6-21, the package properties include the version information.

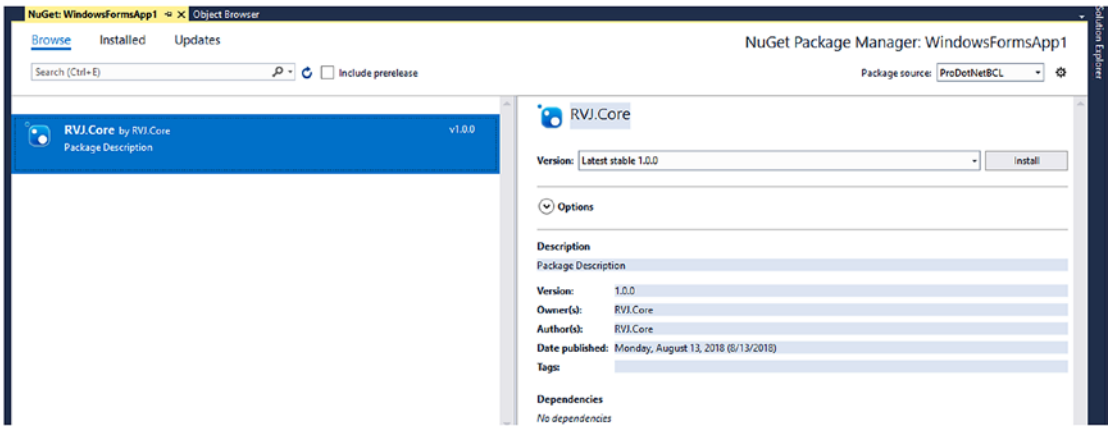


Figure 6-21. *Package properties*

If we choose to install, we have options like latest or other versions, if available. As shown in Figure 6-22, after installing our project, the assemblies are available for use in our project, and they appear on the References list of the project. As shown in Listing 6-17, if we open the file `packages.config` that is created the first time a NuGet packages is imported, we can see that it is an XML file with a different extension. But, every element has a .NET type that can be used to manipulate the content in a specialized manner.

Listing 6-17. Content of a Basic .nupkg Package File

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="RVJ.Core" version="1.0.0" targetFramework="net472" />
</packages>
```

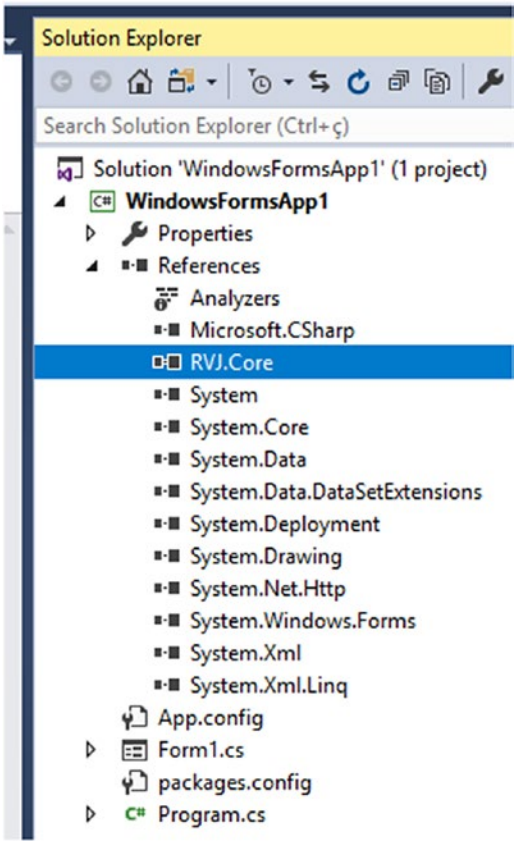


Figure 6-22. *References with NuGet package content and packages.config file*

APPENDIX

Advanced Operations

This appendix will cover the peculiarities of the execution system when dealing with class libraries. The appendix will also talk about decisions we need to make when using native programming languages and managed programming languages.

Unmanaged Code and Managed Code

Most developers already know the fundamentals of how the C and C++ programming languages work with memory blocks. These buffers are viewed and manipulated in code as a sequence of positions. Random access memory (RAM) is organized into bytes. The first byte is at the physical memory address of 0, and the numbering follows consecutively to the last byte, which is in the last physical memory address N-1. Blocks are used to facilitate the navigation and manipulation of bytes by hardware and programming languages.

Each block should have a minimum number and a maximum number of bytes determined by the hardware and supported in its instructions in the Assembly programming language. In programming, instead of dealing with individual bit blocks, the code is organized into blocks of bytes, where each byte is 8 bits in size. For example, when we have a set with two bytes at a time, we have 16-bit units on each block. When we have a set with four bytes at a time, we have units of 32 bits on each block. When we have a set with 8 bytes at a time, we have units of 64 bits on each block, and so on. As already presented, in memory each byte receives a single numeric value, which is its physical address in memory. Well, when it is necessary to access the information in the physical memory (RAM), it is necessary to inform the address. This address in physical memory indicates the location of the byte. The bytes in the addresses in physical memory can indicate any part of the information. This interpretation depends on how the software is designed to handle these bytes. But it is not the software that accesses the physical memory address; the software specifies only the required memory address, and

R. Villela, *Pro .NET Framework with the Base Class Library*, https://doi.org/10.1007/978-1-4842-4191-2

the hardware accesses the byte at the address in physical memory. So, once we access the memory address where the information is, how do we find out the size? The total size of the information, in bytes, determines at which memory address it begins and at which memory address it ends. This is the logic that the byte set organization provides, that is, the beginning, middle, and end.

Moving individual bits from start to finish is pointless when we know where the block starts and where it ends. In fact, moving bit by bit, in addition to being unfeasible in terms of performance, is impractical in terms of energy consumption.

At this point, the pointer-type properties indicate a location. We know that the C and C++ programming languages do not have the concept of arrays as part of their built-in types. So, the syntactic aspects are available as a way to reduce complexity when describing the manipulation of blocks of memory. What really happens is a manipulation via a fundamental built-in type pointer.

As we know, a pointer is a composite type; that is, it works with other types so that it can be effective. The pointer type stores an unsigned integer value that indicates a valid address in memory.

Moving from a memory address to another memory address is performed according to the pointer size. If the pointer has 32 bits, the expressions `(varPointer + 1)`, `varPointer++`, and `++varPointer` all mean advancing 4 bytes from the current position indicated by the pointer. The expressions `(varPointer - 1)`, `varPointer--`, and `--varPointer` all mean go back 4 bytes from the current position indicated by pointer. If the pointer has 64 bits of size, the expressions `(varPointer + 1)`, `varPointer++`, and `++varPointer` mean advance 8 bytes from the current position indicated by the pointer. The drive occurs in "jumps" of N bytes. This size is determined by the hardware architecture for which the application has been designed, 32-bit or 64-bit, for example. On a 32-bit processor, a pointer has a 32-bit size. On a 64-bit processor, a pointer has a 64-bit size. The terms *native pointer size* and *natural pointer size* identify this size according to the processor hardware architecture. Another peculiar aspect of the managed environment is the initialization. Let's review aspects of the initialization using the C++ programming language.

Declaring and Initializing a Variable of Type Pointer

When declaring a variable of type pointer and not initializing, the variable does not point to a valid address in memory. If we try to use this noninitialized variable in some expression, a build or run error occurs. Uninitialized pointers or invalid values are the source of many problems due to lack of stability and security loopholes of the environment. In the statement we can also perform initialization and assign the null value or a valid address in memory. When we use the null value, we indicate that the pointer variable does not store any valid memory address. When we do not assign a valid memory address in the declaration/startup, it is recommended that we always assign the null value to avoid errors and potential security loopholes.

The null value is also used in functions for memory management to indicate when an operation has failed. For example, the `malloc()` function returns the null value (`NULL` in standard C), `nullptr` (in standard C++), or `__nullptr` in MSVC) when the memory allocation attempt fails. Any of the following initializations assigns the null value to the variable of the pointer type. When we try to access the value in a variable of type pointer with the null value, an error occurs. Accessing the value (object) stored in the memory address assigned to the pointer requires the use of the indirection operator (*), and when the pointer variable is not initialized or initialized with the null value, attempting to use the pointer in this manner causes an exception/error to be thrown as a read access violation. Unfortunately, this type of error is quite common and generates many problems, and libraries with specialized features and tools such as compilers use several techniques to try to reduce the occurrence of this type of error. In the coding phase, code analyzers seek to identify potential occurrences and produce visual alerts and reports so that teams can perform proper checks and necessary fixes.

To capture eventualities, programming techniques such as exception management and error management are provided by the C++ programming language, the execution environment, and the Microsoft Windows operating system. In addition, specialized environments such as the CLR platform provide a managed environment. That is, the environment's infrastructure absorbs part of the responsibility required to ensure greater stability and security for applications and provides key facilities such as automatic memory management. In a managed environment like the CLR, the initialization can be automatically performed by the execution environment when certain CIL instructions are explicitly used when the variables are declared.

When a method is defined, we should declare local variables too. The `.locals` directive is used to declare one or more local variables and, when used in conjunction with the optional `init` keyword, causes the variables to be initialized by the execution environment with their default values. In other words, a variable of a reference type is initialized with a null value, and a variable of a value type is zeroed out. See Listing A-1.

Listing A-1. Explicitly Initializing Pointer Variables with a Null Value (Example in the C++ Programming Language)

```
#include <cstdio>
#include <cstdlib>
#include <cstdint>

using namespace std;

int wmain() {

      uint32_t* pointerToN {};
      // uint32_t* pointerToN = nullptr;
      // uint32_t* pointerToN = __nullptr;
      // uint32_t* pointerToN = {};

      /* Displays the zero value that indicates nullptr value. */
      wprintf_s( L"Pointer with value __nullptr (C++) / nullptr (C++):
      0x%p\n", pointerToN );

      /* Displays the memory address of the variable.*/
      wprintf_s( L"Address of pointer variable: %#0x\n", ( ( uint32_t )
      &pointerToN ) );

      return 0;

};
```

In the Listing A-2 and Listing A-3, we are declaring a .NET `System.Array` variable and assigning an instance of `System.Array` for the `System.Int32` value type.

Listing A-2. Declaring Variables with Explicit Initialization (Example in C++/CLI)

```
using namespace System;

int main() {

      ::cli::array<Int32>^ myArray = gcnew  ::cli::array<Int32>( 10i32 );

      return 0i32;
};
```

Listing A-3. Excerpt of the CIL for Listing A-2 Showing Explicit Initialization (Example in CLI)

```
  .method assembly static int32 modopt([mscorlib]System.Runtime.
  CompilerServices.CallConvCdecl)
        main() cil managed
{
  .vtentry 1 : 1
  // Code size       16 (0x10)
  .maxstack  1
  .locals ([0] int32 V_0,
           [1] int32[] myArray)
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.0
  IL_0002:  ldnull
  IL_0003:  stloc.1
  IL_0004:  ldc.i4.s   10
  IL_0006:  newarr     [mscorlib]System.Int32
  IL_000b:  stloc.1
  IL_000c:  ldc.i4.0
  IL_000d:  stloc.0
  IL_000e:  ldloc.0
  IL_000f:  ret
} // end of method 'Global Functions'::main
```

In the Listing A-4 and Listing A-5, we have the same sequence of code but with the C# programming language used for declaring a variable of the `System.Array` abstract reference type.

Listing A-4. Declaring Variables with Implicit Initialization with Default Values (Example in C#)

```
using System;

namespace ConsoleApp1
{
    class Program
    {
        static void Main()
        {
            Int32[] myArray = new Int32[10];

        }
    }
}
```

Listing A-5. Excerpt of the CIL for Listing A-4 Showing Explicit Initialization (Example in CLI)

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       10 (0xa)
  .maxstack  1
  .locals init ([0] int32[] myArray)
  IL_0000:  nop
  IL_0001:  ldc.i4.s   10
  IL_0003:  newarr     [mscorlib]System.Int32
  IL_0008:  stloc.0
  IL_0009:  ret
} // end of method Program::Main
```

When using programming languages for managed execution environments like CLR, we must be aware that the compilers can or cannot use the optional `init` keyword with the `.locals` directive. Even when using the `.locals` directive without the optional `init` keyword, compilers for specific languages can opt to "initialize by itself with default values" in some cases before assigning a valid value like an instance of a managed type. For example, C++/CLI is one of these managed programming languages that does this, as we have seen. This helps prevents situations where the variable was not explicitly initialized on declaration and is used without a proper instance assigned.

In this case, the first attempt to use the variable causes an exception, which is (or derived from) `NullReferenceException` in the case of the CLR. Within C++/CLI, for example, when compiling for optimization and for release, the C++/CLI does not emit the `ldnull` instruction for every situation. The compiler assumes that we know what we are doing. This is a relevante point for people who work with C++ programming language and Assembly programming language, for example.

Listing A-6 shows an explicit initialization with the `ldnull` managed instruction generated by the compiler. But with optimizations and for release, this will not happen, and the first attempt will throw `NullReferenceException` (or some error derived from it), as shown in Listing A-7.

Listing A-6. Explicit Initialization (Example in C++/CLI)

```
using namespace System;

int main() {

      ::cli::array<Int32>^ myArray;

      Array::Clear( myArray, {}, myArray->Length );

      return 0i32;
};
```

Listing A-7. Excerpt of the CIL for Listing A-6 Showing Explicit Initialization (Example in CLI)

```
.method assembly static int32 modopt([mscorlib]System.Runtime.
CompilerServices.CallConvCdecl)
        main() cil managed
{
  .vtentry 1 : 1
  // Code size       19 (0x13)
  .maxstack  3
  .locals ([0] int32[] myArray,
           [1] int32 V_1)
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.1
  IL_0002:  ldnull
  IL_0003:  stloc.0
  IL_0004:  ldnull
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  ldc.i4.0
  IL_0008:  ldloc.0
  IL_0009:  ldlen
  IL_000a:  call       void [mscorlib]System.Array::Clear(class [mscorlib]
                                                          System.Array,
                                                          int32, int32)
  IL_000f:  ldc.i4.0
  IL_0010:  stloc.1
  IL_0011:  ldloc.1
  IL_0012:  ret
} // end of method 'Global Functions'::main
```

Using Managed Types for Array Manipulation

When using vector arrays and nonvector arrays, the VES treats each one of these managed types differently internally. First, a vector array is template (native) based, while a nonvector array is a typical managed type, more specifically an abstract reference type: the `System.Array` class type. As we can remember from the C++ programming language template, once a type (template) is instantiated, we cannot change that type. Therefore, as part of its optimization, the VES has implemented internally a specific set of template definitions of vector arrays for the set of fundamental built-in types. That means when we declare a vector array for a fundamental built-in type like `int32` (`System.Int32` in the BCL core set), the VES has a native template declared for this type. We also cannot directly create derivations of the `System.Array` abstract reference type; only the VES has this capability.

The CIL is also adapted for work with vector arrays (or just *vectors*) and nonvector arrays (or just *arrays*). When we declare a vector array like in Listing A-8 and Listing A-9, the declaration follows a typical syntax using square brackets ([]), and the instantiation occurs by a special intermediate instruction, `newarr`.

Listing A-8. Using a Vector Array (Example in C# and the Generated CIL)

```
/* C# programming language. */

Int32[] numbers = new Int32[ 10 ];
String[] names = new String[ 10 ];
/* CIL - Common Intermediate Language (from C# code). */

.locals init ([0] int32[] numbers, [1] string[] names,...)

  IL_0001:  ldc.i4.s   10
  IL_0003:  newarr     [mscorlib]System.Int32
  IL_0008:  stloc.0
  IL_0009:  ldc.i4.s   10
  IL_000b:  newarr     [mscorlib]System.String
  IL_0010:  stloc.1
```

Listing A-9. Using a Vector Array (Example in C++/CLI and the Generated CIL)

```
::cli::array<Int32>^ numbers{ 10i32 };
::cli::array<String^>^ names{ 10i32 };

/* OR */

::cli::array<Int32>^ numbers = gcnew ::cli::array<Int32>( 10i32 );
::cli::array<String^>^ names = gcnew ::cli::array<String^>( 10i32 );

/* CIL - Common Intermediate Language. */

.locals ([0] class int32 numbers,[1] string[] names,... )

  IL_0000:  ldnull
  IL_0001:  stloc.s    numbers
  IL_0003:  ldnull
  IL_0039:  ldc.i4.s   10
  IL_003b:  newarr     [mscorlib]System.Int32
  IL_0040:  stloc.s    numbers
  IL_0042:  ldc.i4.s   10
  IL_0044:  newarr     [mscorlib]System.String
  IL_0049:  stloc.s    names
```

As shown in Listing A-10 and Listing A-11, if we are using `System.Array` to create instances of `System.Int32` and `System.String`, we are describing for the VES the new type for derivation from the `System.Array` abstract reference type.

Listing A-10. Using a Nonvector Array (System.Array) (Example in C++/CLI and the Generated CIL)

```
/* C++/CLI (Common Language Infrastructure) projection. */

Array^ numbers{ Array::CreateInstance( Int32::typeid, 10i32 ) };
Array^ names{ Array::CreateInstance( String::typeid, 10i32 ) };

/* CIL - Common Intermediate Language. */

  .locals ( [0] class [mscorlib]System.Array numbers,   [1] class
  [mscorlib]System.Array names,...)
```

```
  IL_0000:  ldnull
  IL_0001:  stloc.s    numbers
  IL_0003:  ldnull
  IL_0004:  stloc.s    names
  IL_0039:  ldtoken    [mscorlib]System.Int32
  IL_003e:  call       class [mscorlib]System.Type [mscorlib]System.Type.
  GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_0043:  ldc.i4.s   10
  IL_0045:  call       class [mscorlib]System.Array [mscorlib]System.Array.
  CreateInstance(class [mscorlib]System.Type, int32)
  IL_004a:  stloc.s    numbers
  IL_004c:  ldtoken    [mscorlib]System.String
  IL_0051:  call       class [mscorlib]System.Type [mscorlib]System.Type.
  GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_0056:  ldc.i4.s   10
  IL_0058:  call       class [mscorlib]System.Array [mscorlib]System.Array.
  CreateInstance(class [mscorlib]System.Type, int32)
  IL_005d:  stloc.s    names
```

Listing A-11. Using a Nonvector Array (System.Array) (Example in C# and the Generated CIL)

```
Array numbers = Array.CreateInstance( typeof( Int32 ), 10 );
Array names = Array.CreateInstance( typeof( String ), 10 );

/* CIL - Common Intermediate Language. */

  .locals init ([0] class [mscorlib]System.Array numbers,
           [1] class [mscorlib]System.Array names, ...)

  IL_0001:  ldtoken    [mscorlib]System.Int32
  IL_0006:  call       class [mscorlib]System.Type [mscorlib]System.Type.
  GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_000b:  ldc.i4.s   10
  IL_000d:  call       class [mscorlib]System.Array [mscorlib]System.Array.
  CreateInstance(class [mscorlib]System.Type, int32)
```

```
IL_0012:  stloc.0
IL_0013:  ldtoken    [mscorlib]System.String
IL_0018:  call       class [mscorlib]System.Type [mscorlib]System.Type.
GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
IL_001d:  ldc.i4.s   10
IL_001f:  call       class [mscorlib]System.Array [mscorlib]System.Array.
CreateInstance(class [mscorlib]System.Type,  int32)
IL_0024:  stloc.1
```

When we are working with a vector array or nonvector array, we should be aware that once an instance is created, the size cannot be changed. That is, if we create an instance of `System.Array` and use the `Array.Resize<T>()` static generic method, internally the implementation of the method creates a new instance of a vector array, and using the `Array.Copy()` static method copies all items from the original array instance to the new array instance. Also, the `Array.Copy()` method returns a reference for the new array instance. But the `Array.Copy()` method does not perform the task; the method calls an internal `static extern` method that does the copying between buffers, as shown in Listing A-12 and Listing A-13. The `MethodImplAttribute` reference type with the enum value `MethodImplOptions.InternalCall` means that the method is implemented as part of the VES, and in this case the method is implemented in the C++ programming language.

Listing A-12. Excerpt of the Array.Copy() Public Method

```
public static void Copy(Array sourceArray, Array destinationArray, int
length)
{

if (sourceArray == null) throw new ArgumentNullException("sourceArray");

if (destinationArray == null)
throw new ArgumentNullException("destinationArray");

Copy(sourceArray, sourceArray.GetLowerBound(0), destinationArray,
destinationArray.GetLowerBound(0), length, false);

}
```

Listing A-13. Internal (Execution Environment Implemented) Array.Copy() Method

```
[MethodImplAttribute(MethodImplOptions.InternalCall)]
internal static extern void Copy(Array sourceArray, int sourceIndex, Array
destinationArray, int destinationIndex, int length, bool reliable);
```

The Importance of Vector Arrays and Nonvector Arrays

`System.Array` is an abstract class (reference type), so the `System.Array` types form a hierarchy because all array types that we declare inherit from `System.Array`. This means that when we use the `Array.CreateInstance()` static method, we are informing the VES that it should create a new class type (reference type) derived from the `System.Array` abstract class type (reference type). In Listing A-14, if we use `arr00.GetType()`, and in Listing A-15, if we use `arr00->GetType()`, the result is the name of the new managed type derived from the `System.Array` abstract class type (reference type), which in this example is `System.Int32[]`. If we change the informed managed type for the `Array.CreateInstance()` static method, the nonvector array derived will be from that managed type, for example, `Array::CreateInstace( String::typeid, 10i32 )` in Listing A-15 and `Array.CreateInstance( String.GetType(), 10 )` in Listing A-14. We have a `System.String[]` array class, and the same is true even for custom types. When we use the equality (==) operator or `[AnyType]->Equals()` instance method to compare the `arr0` and `arr01` instances, as in Listing A-14 and Listing A-15, the result is `FALSE` because the standard implementation for equality comparison is used. In other words, `System.Array.Equals()` does not override the method and use the inherited `Object.Equals()` instance method. But when we do an equality operation using this expression (`arr00.GetType() == arr01.GetType()`) in Listing A-14 and (`arr00->GetType() == arr01->GetType()`) in Listing A-15, the result is true for both. This is because even using the `Array.CreateInstance()` static method twice with the same argument values, the VES does not create two (distinct) derivations. In fact, for the fundamental built-in managed type, no derivations are created; the `System.Int32[]` vector array class is used in this case.

Listing A-14. Special Treatment of the VES for Vector Arrays and Nonvector Arrays (Example in C# and Excerpt in CIL)

```
Array arr00 = Array.CreateInstance( typeof( Int32 ), 10 );
Array arr01 = Array.CreateInstance( typeof( Int32 ), 10 );

Int32[] arr02 = new Int32[ 10 ];

Console.WriteLine( ( arr00 == arr01 ).ToString() );
Console.WriteLine( ( arr00 == arr02 ).ToString() );

Console.WriteLine( ( arr00.Equals( arr01 ) ).ToString() );
Console.WriteLine( ( arr00.Equals( arr02 ) ).ToString() );

Console.WriteLine( arr00.GetType().ToString() );

Console.WriteLine( ( arr00.GetType() == arr01.GetType() ).ToString() );

/* CIL - Common Intermediate Language. */

  .locals init ([0] class [mscorlib]System.Array arr00,
           [1] class [mscorlib]System.Array arr01, [2] int32[] arr02,...)
IL_0000:  nop
IL_0001:  ldtoken    [mscorlib]System.Int32
IL_0006:  call       class [mscorlib]System.Type [mscorlib]System.Type.
GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
IL_000b:  ldc.i4.s   10
IL_000d:  call       class [mscorlib]System.Array [mscorlib]System.Array.
CreateInstance(class [mscorlib]System.Type,int32)
IL_0012:  stloc.0
IL_0013:  ldtoken    [mscorlib]System.Int32
IL_0018:  call       class [mscorlib]System.Type [mscorlib]System.Type.
GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
IL_001d:  ldc.i4.s   10
IL_001f:  call       class [mscorlib]System.Array [mscorlib]System.Array.
CreateInstance(class [mscorlib]System.Type,int32)
IL_0024:  stloc.1
IL_0025:  ldc.i4.s   10
IL_0027:  newarr     [mscorlib]System.Int32
IL_002c:  stloc.2
```

Listing A-15. Special Treatment of the VES for Vector Arrays and Nonvector Arrays (Example in C# and Excerpt in CIL)

```
Array^ arr00{ Array::CreateInstance( Int32::typeid, 10i32 ) };
Array^ arr01{ Array::CreateInstance( Int32::typeid, 10i32 ) };

::cli::array<Int32>^ arr02 = gcnew ::cli::array<Int32>( 10i32 );

Console::WriteLine( ( arr00 == arr01 ).ToString( ) );
Console::WriteLine( ( arr00 == arr02 ).ToString( ) );

Console::WriteLine( ( arr00->Equals( arr01 ) ).ToString() );
Console::WriteLine( ( arr00->Equals( arr02 ) ).ToString() );

Console::WriteLine( arr00->GetType()->ToString() );

Console::WriteLine( ( arr00->GetType( ) == arr01->GetType( ) ).ToString( )
);

/* CIL - Common Intermediate Language. */

  .locals ( [0] class [mscorlib]System.Array arr00,
            [1] class [mscorlib]System.Array arr01, [2] int32[] arr02, ...)

  IL_0000:  ldnull
  IL_0001:  stloc.s    arr00
  IL_0003:  ldnull
  IL_0004:  stloc.s    arr01
  IL_0006:  ldnull
  IL_0007:  stloc.2
  IL_0013:  ldtoken    [mscorlib]System.Int32
  IL_0018:  call       class [mscorlib]System.Type [mscorlib]System.Type.
  GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_001d:  ldc.i4.s   10
  IL_001f:  call       class [mscorlib]System.Array [mscorlib]System.Array.
  CreateInstance(class [mscorlib]System.Type, int32)
  IL_0024:  stloc.s    arr00
  IL_0026:  ldtoken    [mscorlib]System.Int32
  IL_002b:  call       class [mscorlib]System.Type [mscorlib]System.Type.
  GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
```

```
  IL_0030:  ldc.i4.s   10
  IL_0032:  call       class [mscorlib]System.Array [mscorlib]System.Array.
  CreateInstance(class [mscorlib]System.Type,int32)
  IL_0037:  stloc.s    arr01
  IL_003a:  ldc.i4.s   10
  IL_003c:  newarr     [mscorlib]System.Int32
  IL_0041:  stloc.s    arr02
```

Well, when we do a comparison using an equality operation with the expression `arr02->GetType( ) == arr00->GetType( )`, the result is true. However, as we already talked about, the **vector array** and **nonvector** array are distinct types as defined by the **CTS**, represented in **metadata** data structures, and put alive by **VES**, so how is that possible?

Internally the vector array is declared as a template (C++ programming language) and implemented by the VES. For the fundamental built-in data types as declared in CTS, like `int32` (`System.Int32` value type in the BCL core set), the runtime infrastructure of the execution system has a set of template instantiations (C++ programming language), as shown in Listing A-18. The argument type for template instantiations is defined as showed in Listing A-16. When we are working with vector arrays or nonvector arrays, what we have at runtime is an unmanaged internal buffer (on the managed heap) encapsulated and accessed via a few basic managed operations. This is because the majority of infrastructure operations such as instantiation, search, and resizing are made by internal methods. That is, the implementation of these operations is written in the C++ programming language and is part of the implementation of the VES itself, or they have external implementations also written in the C++ programming language with support of array native class types and the runtime infrastructure native types. In the listings, `ArrayHelper.IndexOfUINT8` is an example.

Listing A-16. Excerpt of C++ Files Used by the VES of the CLR

```
typedef __int8              I1;
typedef unsigned __int8     U1;
typedef __int16             I2;
typedef unsigned __int16    U2;
typedef __int32             I4;
typedef unsigned __int32    U4;
```

```
typedef __int64             I8;
typedef unsigned __int64    U8;
typedef float               R4;
typedef double              R8;
```

`ArrayBase` (in the C++ programming language) is the base class type for arrays of any type in the CLR, meaning vector arrays and nonvector arrays. Here we can see one of the public member functions that is used by the array instance to check whether the array is a nonvector array (`System.Array` in the BCL core set), namely, the `ArrayBase.IsMultiDimArray()`. The `ArrayBase.GetMethodTable()` member function is inherited from the `Object` (in the C++ programming language) base class type. In fact, it is an `Object.GetMethodTable()` member function that returns a pointer to one of the data structures of the metadata system of the platform, the `MethodTable` class type. The `MethodTable` (C++ programming language) class type has the `MethodTable.IsArray()` and `MethodTable.IsMultiDimArray()` member functions, for example. The `MethodTable.IsArray()` and `MethodTable.IsMultiDimArray()` member functions use the `MethodTable.GetFlag()` member function with values that indicate whether the instance of array type is a vector array or a nonvector array.

In the Listing A-17 the values of `WFLAGS_HIGH_ENUM` help us to understand the array type used by the execution environment of the managed platform. `WFLAGS_HIGH_ENUM.enum_flag_Category_Array` with the value 0x00080000 identifies an array type, and `WFLAGS_HIGH_ENUM.enum_flag_Category_IfArrayThenSzArray` with the value 0x00020000 is a subcategory of array. When the value `WFLAGS_HIGH_ENUM.enum_flag_Category_Array_Mask` with the value 0x000C0000 is informed to the `MethodTable.GetFlag()` member function, an operation & (bit-by-bit) is performed, and the result should be equal to `WFLAGS_HIGH_ENUM.enum_flag_Category_Array` if the instance is a vector array. See Listing A-17.

Listing A-17. Implementation of the methods that identifies if the current instance is a vector or nonvector and an example of the use by the ArrayBase class

```
class MethodTable {

public:

    // Is this array? Returns false for System.Array (non-vector array).
    inline BOOL IsArray()
    {
        LIMITED_METHOD_DAC_CONTRACT;
```

```
        return GetFlag(enum_flag_Category_Array_Mask) == enum_flag_
        Category_Array;

    }

    inline BOOL IsMultiDimArray()
    {
        LIMITED_METHOD_DAC_CONTRACT;
        PRECONDITION(IsArray());
        return !GetFlag(enum_flag_Category_IfArrayThenSzArray);

    }
};

class ArrayBase : public Object {

public:
    BOOL IsMultiDimArray() const {
        WRAPPER_NO_CONTRACT;
        SUPPORTS_DAC;

        return(GetMethodTable()->IsMultiDimArray());

    }

};
```

This is the template used for all nonvector arrays (nonobject arrays in the terminology of the VM) with a single dimension, that is, vector arrays. Just the member field that stores the internal buffer is shown in Listing A-18, but the template has more members that are not shown here.

Listing A-18. Excerpt of the implementation of the ArrayBase class with the KIND template parameter, that is used to defines the base type of the template instantiation

```
template < class KIND >
class Array : public ArrayBase {
  public:

    KIND          m_Array[1];

};
```

```
typedef Array<I2>   I2Array;
typedef Array<I1>   I1Array;
typedef Array<I4>   I4Array;
typedef Array<I8>   I8Array;
typedef Array<R4>   R4Array;
typedef Array<R8>   R8Array;
typedef Array<U1>   U1Array;
typedef Array<U1>   BOOLArray;
typedef Array<U2>   U2Array;
typedef Array<WCHAR>   CHARArray;
typedef Array<U4>   U4Array;
typedef Array<U8>   U8Array;
```

Listing A-19 shows one of the specialized "index" operations implemented in the helper class `ArrayHelper`. As we can see, the member function is specialized for arrays with elements of `UINT8` type (unsigned `__int8`). Also, the operation uses the `memchr` function (Standard C/C++ Library, Microsoft UCRT, Microsoft CRT), comparing byte by byte.

Listing A-19. Excerpt of the Implementation of the ArrayHelper.IndexOfUINT8() Member Function

```
INT32 ArrayHelper.IndexOfUINT8( UINT8* array, UINT32 index, UINT32 count,
UINT8 value) {

    LIMITED_METHOD_CONTRACT;

    UINT8 * pvalue = (UINT8 *)memchr(array + index, value, count);
    if ( NULL == pvalue ) {
        return -1;
    }
    else {
        return static_cast<INT32>(pvalue - array);
    }
};
```

Using the Array.CreateInstance() Static Unsafe Method

Five of the six signatures of the implementations of the `Array.CreateInstance()` static methods have the **`unsafe`** keyword used in their declaration because the method that really does the work is the `Array.InternalCreate()` static method. This means we cannot waste time by creating inefficient managed code because managed code has limits. When necessary, use native code. Listing A-20 shows excerpts of the `Array.CreateInstance()` method for each one of the six signatures.

Listing A-20. Excerpts of Array.CreateInstance() Method Implementations

```
/*Here is the declaration of the Array.InternalCreate() method that is
internal and part of the VES implementation:*/

[MethodImplAttribute(MethodImplOptions.InternalCall)]

private unsafe static extern Array InternalCreate(void* elementType,int
rank,int *pLengths,int *pLowerBounds);

/*Here is the first implementation:*/

public unsafe static Array CreateInstance(Type elementType, int length)
{
...
RuntimeType t = elementType.UnderlyingSystemType as RuntimeType;
if (t == null)throw new ArgumentException(Environment.
GetResourceString("Arg_MustBeType"),"elementType");
return InternalCreate((void*)t.TypeHandle.Value,1,&length,null);

}

/*Here is the second implementation:*/

public unsafe static Array CreateInstance(Type elementType, int length1,
int length2)
{
...
```

```
if ((object)elementType == null) throw new ArgumentNullException("elementType");
if (length1 < 0 || length2 < 0)
throw new ArgumentOutOfRangeException((length1 < 0 ? "length1" : "length2"),
Environment.GetResourceString("ArgumentOutOfRange_NeedNonNegNum"));
...

RuntimeType t = elementType.UnderlyingSystemType as RuntimeType;
if (t == null)
throw new ArgumentException(Environment.GetResourceString("Arg_MustBeType"),
"elementType");

int* pLengths = stackalloc int[2];
pLengths[0] = length1;
pLengths[1] = length2;
return InternalCreate((void*)t.TypeHandle.Value,2,pLengths,null);
}

/*Here is the third implementation:*/

public unsafe static Array CreateInstance(Type elementType, int length1,
int length2, int length3)
{

if ((object)elementType == null)
throw new ArgumentNullException("elementType");
if (length1 < 0) throw new ArgumentOutOfRangeException("length1",
Environment.GetResourceString("ArgumentOutOfRange_NeedNonNegNum"));
if (length2 < 0) throw new ArgumentOutOfRangeException("length2",
Environment.GetResourceString("ArgumentOutOfRange_NeedNonNegNum"));
if (length3 < 0) throw new ArgumentOutOfRangeException("length3",
Environment.GetResourceString("ArgumentOutOfRange_NeedNonNegNum"));

...
RuntimeType t = elementType.UnderlyingSystemType as RuntimeType;
if (t == null) throw new ArgumentException(Environment.
GetResourceString("Arg_MustBeType"), "elementType");
```

```
int* pLengths = stackalloc int[3];
pLengths[0] = length1;
pLengths[1] = length2;
pLengths[2] = length3;
return InternalCreate((void*)t.TypeHandle.Value,3,pLengths,null);

}

/*Here is the fourth implementation:*/

public unsafe static Array CreateInstance(Type elementType, params int[]
lengths) {
if ((object)elementType == null)
throw new ArgumentNullException("elementType");
if (lengths == null)
throw new ArgumentNullException("lengths");
if (lengths.Length == 0)
throw new ArgumentException(Environment.GetResourceString("Arg_
NeedAtLeast1Rank"));

...

 RuntimeType t = elementType.UnderlyingSystemType as RuntimeType;
 if (t == null)
 throw new ArgumentException(Environment.GetResourceString("Arg_MustBeType"),
"elementType");

...

 for (int i=0;i<lengths.Length;i++)
    if (lengths[i] < 0)
          throw new ArgumentOutOfRangeException("lengths["+i+']',
          Environment.GetResourceString("ArgumentOutOfRange_
          NeedNonNegNum"));

fixed(int* pLengths = lengths)
return InternalCreate((void*)t.TypeHandle.Value,lengths.Length,
pLengths,null);
}
```

```
/*Here is the fifth implementation:*/

public static Array CreateInstance(Type elementType, params long[] lengths)
{
if( lengths == null) {
throw new ArgumentNullException("lengths");
}
            if (lengths.Length == 0)
                throw new ArgumentException(Environment.GetResource
                String("Arg_NeedAtLeast1Rank"));

...
            int[] intLengths = new int[lengths.Length];

            for (int i = 0; i < lengths.Length; ++i)
            {
                long len = lengths[i];
                if (len > Int32.MaxValue || len < Int32.MinValue)
                    throw new ArgumentOutOfRangeException("len",
                    Environment.GetResourceString("ArgumentOutOfRange_
                    HugeArrayNotSupported"));
                intLengths[i] = (int) len;
            }

            return Array.CreateInstance(elementType, intLengths);
}

/*Here is the sixth implementation:*/

public unsafe static Array CreateInstance(Type elementType, int[]
lengths,int[] lowerBounds) {

...
RuntimeType t = elementType.UnderlyingSystemType as RuntimeType;

if (t == null)
throw new ArgumentException(Environment.GetResourceString("Arg_MustBeType"),
"elementType");
```

```
for (int i=0;i<lengths.Length;i++)
   if (lengths[i] < 0)
     throw new ArgumentOutOfRangeException("lengths["+i+']',
     Environment.GetResourceString("ArgumentOutOfRange_NeedNonNegNum"));
   fixed(int* pLengths = lengths)
      fixed(int* pLowerBounds = lowerBounds)
      return InternalCreate((void*)t.TypeHandle.Value,lengths.Length,
      pLengths,pLowerBounds);
}
```

Avoiding the Use of Array.CopyTo() Instance Method

There is nothing wrong with the implementation of the method `Array.CopyTo()`, as shown in Listing A-21. But these instance methods do a check and throw an exception, and we also get one or two extra indirections. We can, and should, avoid this checking made by the implementations of the `Array.CopyTo()` instance methods with good code practice, and with this, also avoid the occurrence and execution time cost of the exceptions and these extra indirections.

Instead, we should use `Array.Copy()`.

Listing A-21. Excerpts of Array.CopyTo() Implementations

```
public void CopyTo(Array array, int index)
{
if (array != null && array.Rank != 1)
throw new ArgumentException(Environment.GetResourceString("Arg_
RankMultiDimNotSupported"));
Array.Copy(this, GetLowerBound(0), array, index, Length);
}
public void CopyTo(Array array, long index)
{
if (index > Int32.MaxValue || index < Int32.MinValue)
                throw new ArgumentOutOfRangeException("index",
                Environment.GetResourceString("ArgumentOutOfRange_HugeArray
                NotSupported"));
```

```
this.CopyTo(array, (int) index);
}
```

Examining the Other Methods of System.Array

The same behaviors apply in the other methods of `System.Array`; that is, the public and nonpublic managed methods have been created to filter certain aspects of arguments values and to avoid unnecessary overload of repetitive validation tasks inside the VES. Avoiding these kinds of tasks, the execution environment becomes much more stable. Just to see how this implementation engineering model is critical, see the following list of some methods of `System.Array` and the elements of interaction with internal methods of the VES:

- `Array.ConstrainedCopy()` uses the `Copy` method implemented as part of the execution engine.
- The method `Array.GetLongLength()` calls the public `Array.GetLength()` method that is implemented as part of the execution engine. We also can call the `Array.GetLength()` method directly from our code.
- The `[ArrayInstance]->Length::get()` (C++/CLI) function member operation is implemented as part of the execution engine, and we can call "directly" from our code.
- Methods that perform search operations such as `Array.IndexOf()` use a private `Array.TrySZIndex()` method that is implemented as part of the execution engine, the VES. So, they have a better performance. But the performance could be affected by a custom implementation of equality operations.

It is important to understand that even when the implementation of the method is marked with `MethodImplOptions.InternalCall`, it is made only with managed code, and it can be designed to be implemented as part of the execution runtime. On the installation of the CLR/.NET Framework/.NET Core and specialized runtimes based on them, a set of managed components is compiled to native code based on the target environment (operating system and hardware platform), and the native binaries (DLLs and EXEs) are generated and are installed on the target environment. When a call is performed to one of these members (methods for example) of a managed type that is part of the execution environment and has a native implementation generated, the native implementation

is called by the runtime. Therefore, we get adequate performance based on the criteria defined by the architecture and engineering implementation of the managed platform for the target environment, that is, the operating system and the hardware platforms.

Optimizations in the VES

If we are designing and implementing components and controls that should use the C++ programming language, we should optimize the VES. What is important is that we do not have to try to reinvent all things via the C# programming language or create mixed code using C++/CLI and the C++ programming language. After optimizing, we can create a layer for .NET interaction with these specialized components and controls written in native code. This layer can be implemented using C++/CLI or the C# programming language, depending on the purpose and necessary resources. When we need to work with managed and unmanaged code, more specifically C++ code, we should create simple applications or class libraries experimenting with base functionalities (operations). Create a simple list of directions like these examples:

- Required data structure: vector array
    - Managed
        - Maybe/should be written in C# programming language
        - Maybe/should be written in C++/CLI
    - Native
        - Maybe/should be written in C++ programming language
        - Maybe/should be written in Assembly programming language
    - Purpose
        - Stores a sequence with a maximum of 500 elements at a time
        - Performs searches on this sequence

If any item is not adequate for what is considered for the application, just remove it from the list, like the use of the Assembly programming language, for example. Define the directions using “Maybe” or “Should,” indicating that certain elements have been defined and others are open to suggestion or discussion. After the basic lists for directions, reaffirm the aspects of data types and data structures considering the

peculiarities of the managed environment or native environment. Create explanations like in these examples:

- Vector array
  - Single-dimensional, zero-based array.
  - Should have a maximum of 500 elements at a time.
  - Used for cache of items in lists used for searches.
  - The cache of items, depending of context, maybe or should be integrated with GUI or non-GUI components/controls.
- Managed environment (CLR in this case)
  - Should use vector array as managed type.
  - Should avoid creating new instances all the time.
  - For fundamental built-in types, the VES has an optimization to declare internal vector arrays using C++ templates. This is an advantage when using CTS fundamental built-in types such as `int32`, `string`, and `char`, for example, and when using custom managed types.
- Native environment (Microsoft Windows in this case)
  - Should use the blocks of memory to store arrays of pointers.
  - Should create a thin layer with a public application API to create a minimum level of isolation of native operating system functions for the manipulation of blocks of memory. Here are examples:
    - `void* Allocation( void* pointer, uint32_t sizeInBytes );`
    - `void* Allocation( void* pointer, uint64_t sizeInBytes );`
    - `bool Release( void* pointer );`
    - `Error_Memory_Operation_Type Check( void* block, void* start, void* end, uint32_t expectedSize );`

    - `Error_Memory_Operation_Type Check( void* block, void* start, void* end, uint64_t expectedSize );`
    - `Error_Memory_Operation_Type (enum)`
- Internally the allocation functions can use Standard C/C++ Library/Microsoft UCRT/Microsoft CRT, Microsoft Windows Heap Functions or even Microsoft Windows virtual functions when necessary.
- Should create this set of functions in the global namespace of the C++ programming language.
- Should encapsulate this set of functions in a basic set of managed types using C++/CLI or the C# programming language.
- The native library should be created with Microsoft Visual C++ (part of Microsoft Visual Studio 2017 17 15.n or more recent) or Intel C++ (Intel Parallel Studio XE 2018 or more recent). Depending of the optimizations, we can use both products.

For example, follow these instructions for creating a native DLL that compiles with Microsoft Visual C++ (the examples use Microsoft Visual Studio 2017 15.n) or Intel C++ (the tutorial uses Intel Parallel Studio XE 2019 but is valid for Intel Parallel Studio XE 2018 RTM, Update 1, Update 2, and Update 3).

First, use the File ➤ New Project menu option and Visual C++ ➤ Dynamic-Link (DLL) template. Enter the name **RVJ.Memory.Management**, as shown in Figure A-1.

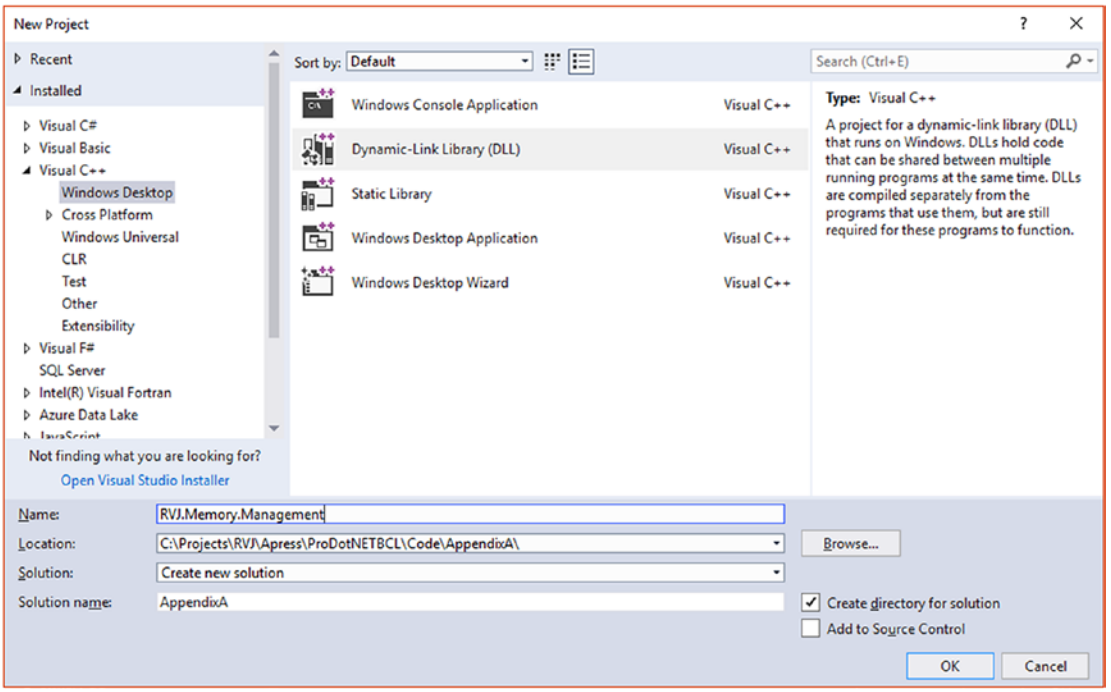


Figure A-1. *New DLL project*

As shown in Figure A-2, the name of solution is AppendixA, and the name of project (not necessarily the name of output binary) is RVJ.Memory.Management.

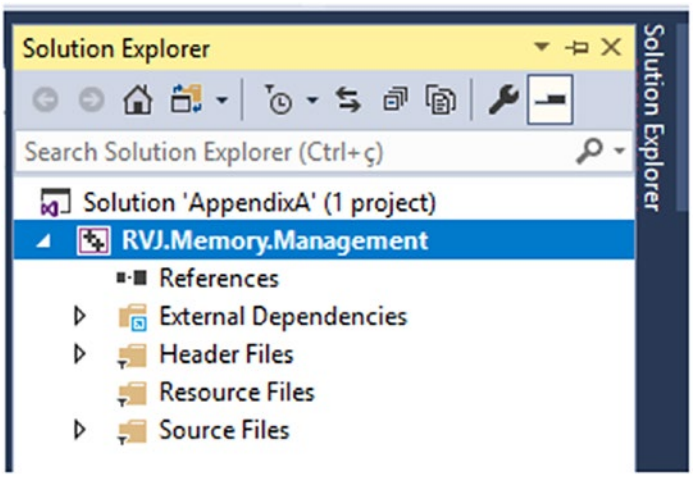


Figure A-2. *Microsoft Visual Studio Solution Explorer with solution and project*

Next, as shown in Figure A-3, if we are thinking of using Intel C++ with the project, we can alternate back and forth between the Intel product by clicking the Microsoft Visual Studio menu and selecting Project ➤ Intel Compiler ➤ Use Intel C++.

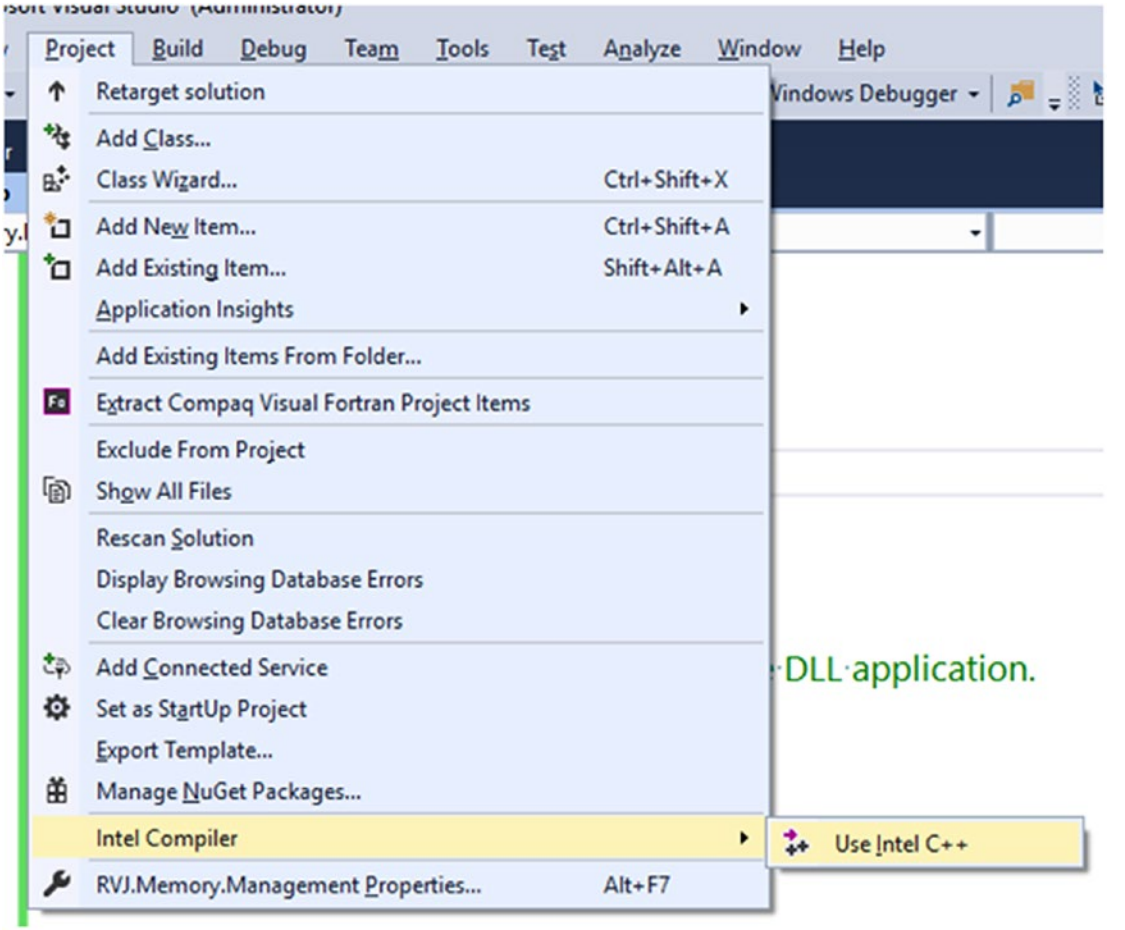


Figure A-3. *Configuring Microsoft Visual Studio for use with Intel C++ tools*

After choosing to change to the Intel C++ tools and configuration, we will see a message box, as shown in Figure A-4.

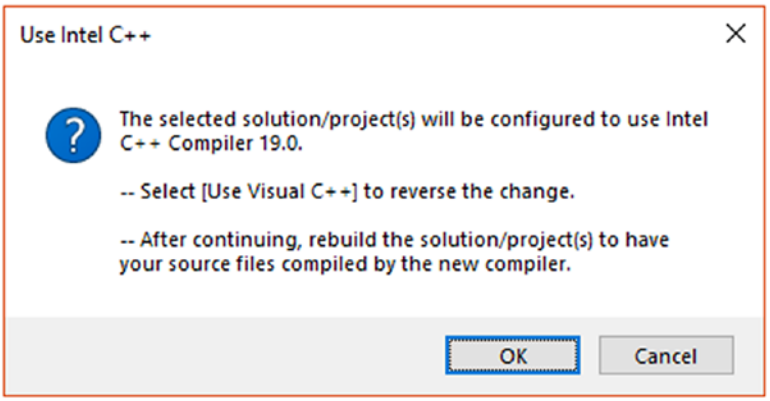


Figure A-4. *Microsoft Visual Studio message alert about the configuration*

After the changes are made, the output window shows some information about what happened, as shown in Figure A-5.

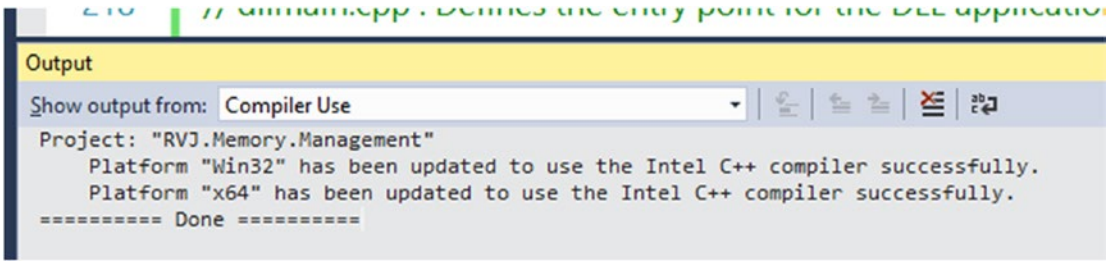


Figure A-5. *Microsoft Visual Studio Output window with messages about the update of configuration*

Figure A-6 shows the Solution Explorer window with the project configured to work with Intel C++.

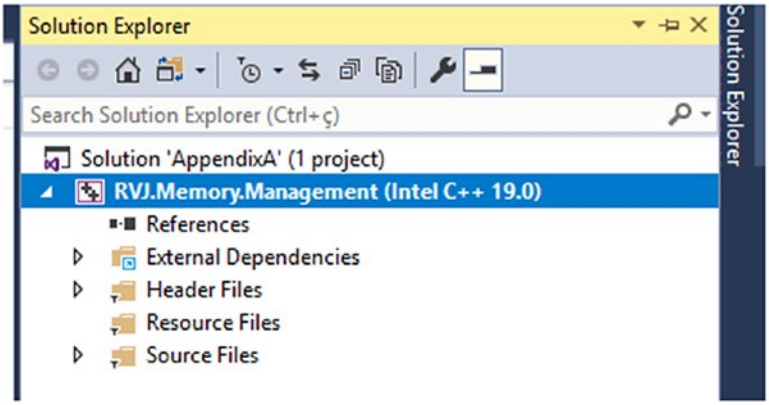


Figure A-6. *Microsoft Visual Studio Solution Explorer with visual notification about the Intel C++ environment tools*

After changing to the Intel C++ environment, Microsoft Visual C++ has new configuration sections that are available for the Intel C++ tools, as shown in Figure A-7.

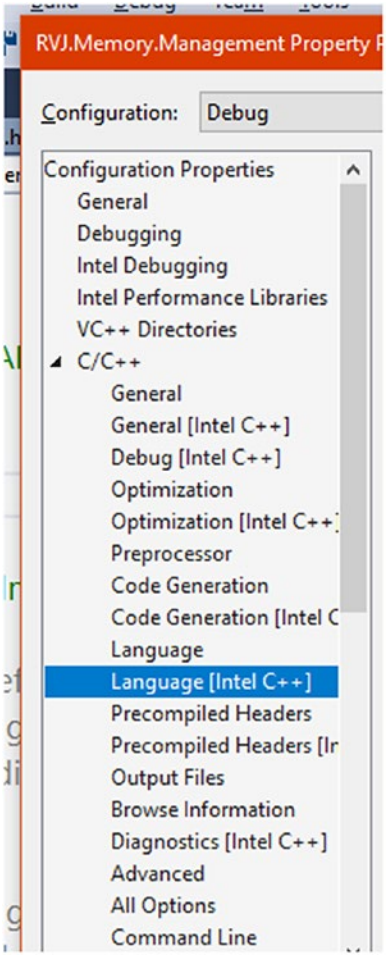


Figure A-7. *Microsoft Visual Studio Project Properties window with sections with specialized configurations for the Intel C++ environment*

Figure A-8 shows some details about one of the configurations of the Intel C++ compiler and possible option conflicts that can occur with the Microsoft Visual C++ compiler.

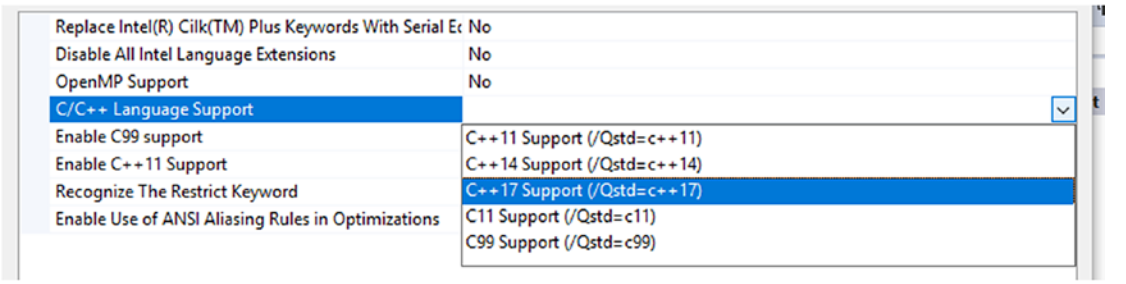


Figure A-8. *Configurations can generate conflicts between C++ environments*

In Figure A-9 we have an example of a conflict that was generated because of a compiler option and how to solve it.

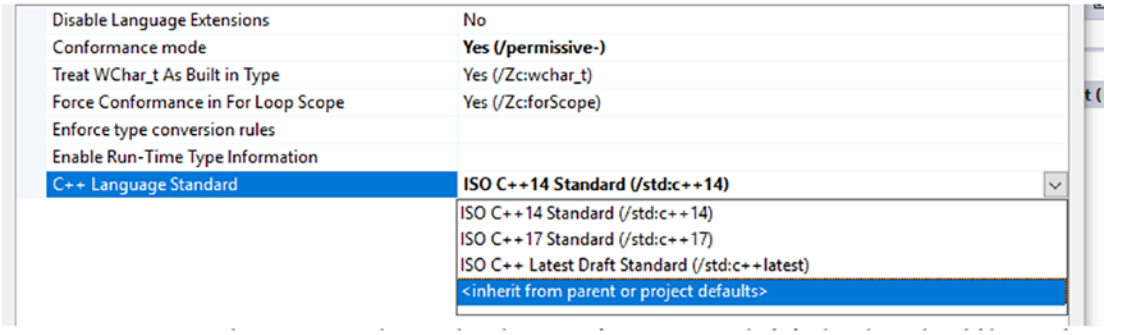


Figure A-9. *When using Intel++ and encountering conflicts, in general, default values should be used for the Microsoft Visual C++ environment and vice versa*

Figure A-10 shows an example of a code adaptation that is necessary when working with two or more compilers on the same code base.

```
#ifdef __INTEL_COMPILER
#pragma warning( disable: 1079; )
#endif
```

Figure A-10. *Adaptations will be necessary for each C++ environment and C++ code, like the use of specific macros or code constructions*

As shown in Figure A-11, when we need to change back to Microsoft Visual C++, we should use the Microsoft Visual Studio menu Project ➤ Intel C++ ➤ Use Visual C++.

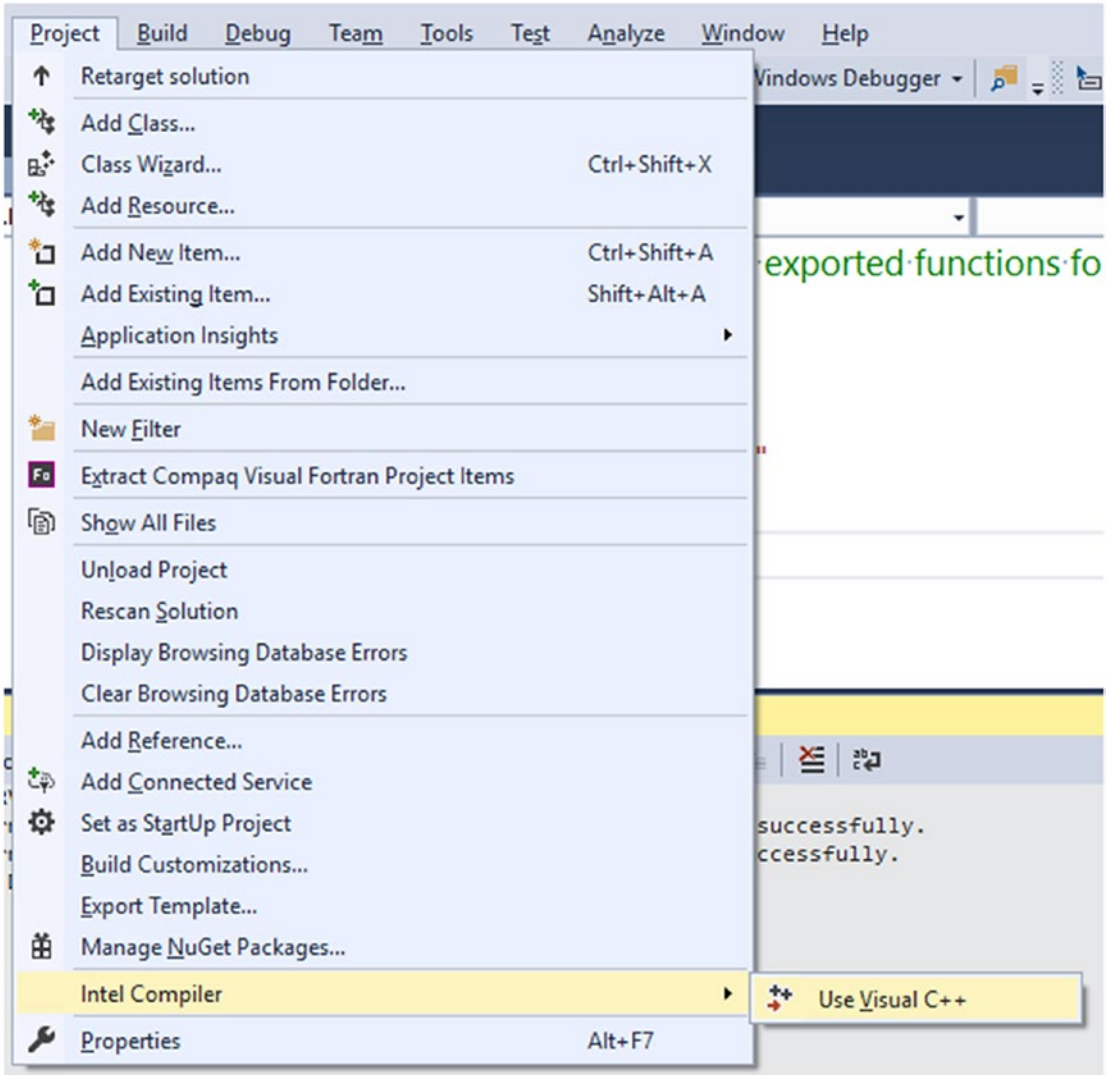


Figure A-11. *Changing back to the Microsoft Visual C++ environment*

After switching to the Visual C++ tools and configuration, a message box is presented, as shown in Figure A-12.

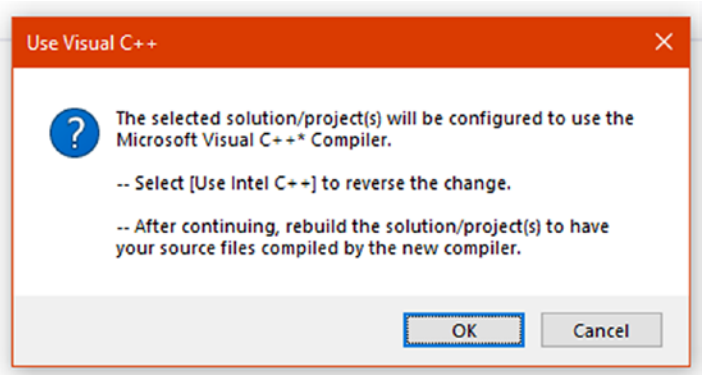


Figure A-12. *Again, an alert about the updating on the C++ environment*

When working with distinct products and environments for development with the C++ programming language, we should pay attention to things that we already know are necessary, as shown in Listing A-22. For example, the C++ code should compile and generate an adequate binary for each target implementation of the operating system. In other words, the same block of code should work or be adapted to work on the C++ programming language development environments.

Listing A-22. Checking for Distinct Operating System Implementations

```
#include <Windows.h>
#include <cstdint>

using namespace std;

#ifdef _WIN32
typedef uint32_t Natural_Size;
#else
typedef uint64_t Natural_Size;
#endif
```

As shown in Listing A-23, for our memory management implementation, if we are using a Microsoft Windows API such as Microsoft heap allocation functions or Standard C/C++ Library/Microsoft UCRT/Microsoft CRT, we can adapt the code as necessary, because the client components (managed or native) do not know about the implementation details, as shown in Figure A-13.

Listing A-23. Example of "Draft" Memory Allocation Function/Deallocation Function

```
bool Allocation( void const* newBuffer, Natural_Size sizeInBytes ) {
      bool result{};
      if ( sizeInBytes > Natural_Size( 0 ) ) {
            void* localBuffer{ ( void* )HeapAlloc( GetProcessHeap(),
            HEAP_ZERO_MEMORY, sizeInBytes ) };
            if ( ( result = GetLastError() == NO_ERROR  ) ) newBuffer =
            localBuffer;
      };
      return result;
};

bool Release( void const* newBuffer ) {
      bool result{ newBuffer != __nullptr };
      //OR
      //BOOL result{ newBuffer != __nullptr };
      if ( result  || ( result = HeapFree( GetProcessHeap(), {}, ( ( LPVOID )
      newBuffer ) ) ) ) {
            if ( ( result = GetLastError() == NO_ERROR ) ) { /* do
            something. */ }
            else { /* do something. */ };
      };
      return result;
};
```

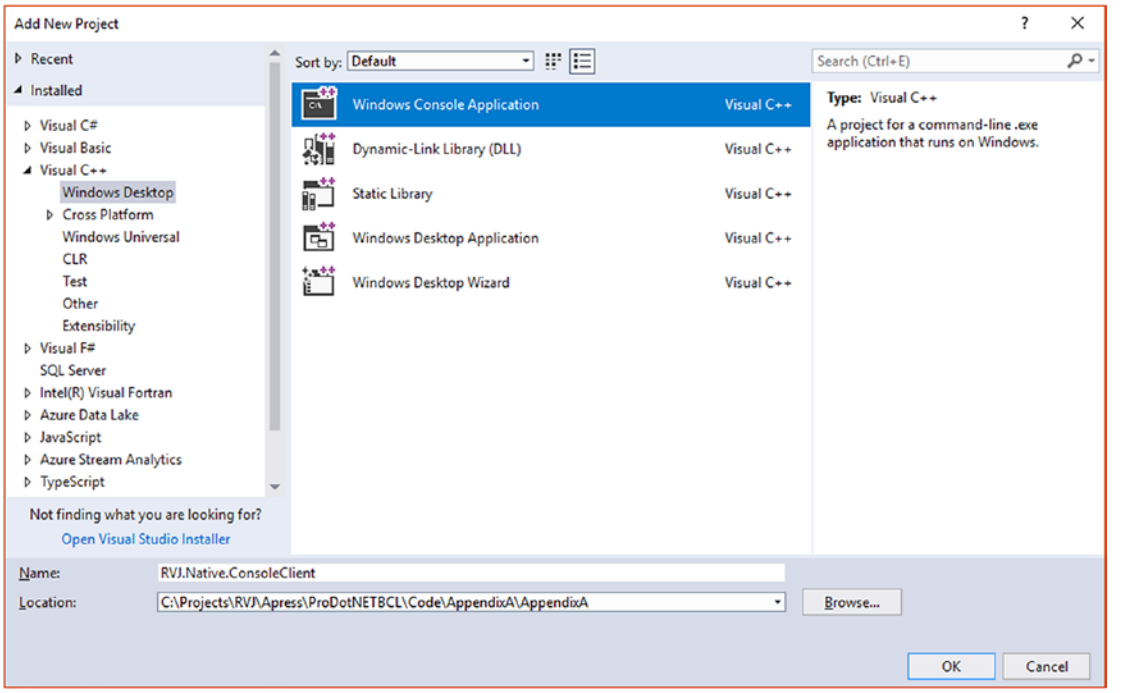


Figure A-13. *Adding a client project with the name RVJ.Native.ConsoleClient to the solution*

As shown in Listing A-24, now with the infrastructure DLLs ready, we should experiment with how the APIs should be used on the client code, doing so in C++ programming language initially. An adequate model for the API at the client code level is that it is objective but without losing the flexibility. For example, if we require more than one implementation, we should review whether a new signature and implementation are really necessary for "just" one parameter or whether it is adequate to include the new parameter on the existing function and adapt the implementation to do checks for the required conditions associated with the argument value.

Listing A-24. Using APIs on Client Code

```
typedef char16_t BaseType;
constexpr uint32_t BaseTypeSize{ sizeof( BaseType ) };
constexpr uint32_t BufferSize{ 100ui32 };
constexpr uint32_t SizeInBytes{ BufferSize << BaseTypeSize  };
```

```
char16_t* newBuffer{ };

      if ( Allocation( newBuffer, SizeInBytes ) ) {

            WriteMessage( u"newBuffer allocated!" );

            if ( Release( newBuffer ) ) WriteMessage( u"newBuffer
            deallocated!" );
            else WriteMessage( u"Houston we have a problem!" );

      };

      newBuffer = nullptr;
```

After the fundamental infrastructure native code, we should consider whether the layer using the C# programming language with P/Invoke is more adequate to access the fundamental infrastructure written in native code or if we should write all code of this new layer using C++/CLI. This is not because of the "it just works" support in C++/CLI and the C++ programming language but because we already have decided that we will use the P/Invoke mechanism just to use one base programming language or extension.

We must remember that we are talking about the infrastructure runtime layer of contextualized libraries and not about the code for graphical user interface (GUI environments produced by specialized visual designers of Microsoft Visual Studio. One typical element of using a managed language like the C# programming language is that "extra" members and types are sometimes and unnecessarily included because of the implementation model used by libraries like the BCL (core set) or even the .NET FCL (complete set). We must be aware of the capabilities of these native programming languages, projections, and extensions, and of the capabilities of the managed programming languages like the C# programming language. Here we have a sequence of elements that if and when necessary could be structured as not public, because only the name of the member suggests that it "should" be used or that it is "critical," which is not the reality.

Deciding Which APIs to Use

One aspect is that we do not need to use all type members that we like. Object-oriented programming, interfaces (component-based knowledge and models), and managed environments all have an exceptional amount of details, and the interaction with the underlying operating system platform and hardware platform has a direct influence on the decisions these software development platforms and programming models make.

See these comments about a few members.

- `Array.IsFixedSize`
  - All arrays are of fixed size, so there is no reason to realize this check. The `System.Array` implements `IsFixedSize` property because it is part of the `System.Collections.IList` interface and the `System.Array` implements the `System.Collections.IList` interface. The implementation of `IsFixedSize` property by `System.Array` always return true, as showed in the example in Listing A-25.

Listing A-25. Excerpt of System.Array Implementation from System.Collections.IList

```
public bool IsFixedSize {
get { return true; }
}
```

- `Array.Count`
  - This member is for an explicit implementation of the `System.ICollection` interface. An example of the implementation of this member is showed in Listing A-26.

Listing A-26. Excerpt of the System.Array Implementation from ICollection.Count

```
// Number of elements in the Array.
int ICollection.Count
        { get { return Length; } }
```

- `Array.IsReadOnly`
  - All arrays are of fixed size, and read-only is always false, so there is no reason to realize this check. The implementation of the member is shown in Listing A-27.
  - When cast for `System.Collections.IList`, the `IsReadOnly` property keeps the behavior, that is, returns `false`. When cast for `System.Collections.Generic.IList<Int32>`, the `IsReadOnly` property returns true. An example of this behavior is shown in Listing A-28 for both C# and C++/CLI.

Listing A-27. Excerpt of System.Array Implementation from System.Collections.Generic.IList<T>

```
// Is this Array read-only?
public bool IsReadOnly
        { get { return false; } }
```

Listing A-28. IsReadOnly Behavior

```
/* C# programming language. */

Int32[] numbers = { -1, -2, -3, -4, -5, -6, -7, 0, 1, 2, 3, 4, 5, 6, 7 };

Console.WriteLine( "Arrray.IsReadOnly: {0}", numbers.IsReadOnly.ToString() );
Console.WriteLine( "System.Collections.IList.IsReadOnly: {0}", ( ( System.
Collections.IList ) numbers ).IsReadOnly.ToString() );
```

```
Console.WriteLine( "System.Collections.Generic.IList<Int32>.IsReadOnly:
{0}", ( ( System.Collections.Generic.IList<Int32> ) numbers ).IsReadOnly.
ToString() );

/* C++/CLI (Common Language Infrastructure). */

::cli::array<Int32>^ numbers = { -1, -2, -3, -4, -5, -6, -7, 0, 1, 2, 3, 4,
5, 6, 7 };

Console::WriteLine(  "Array->IsReadOnly: {0}", numbers->IsReadOnly.
ToString() );
Console::WriteLine(  "System::Collections::IList::IsReadOnly: {0}",
( ( System::Collections::IList^ ) numbers )->IsReadOnly.ToString( ) );
Console::WriteLine(  "System::Collections::Generic::IList<Int32>::IsRea
dOnly: {0}", ( ( System::Collections::Generic::IList<Int32>^ ) numbers
)->IsReadOnly.ToString( ) );
```

- `Array.Length`
  - `Array.Length.get()` is fast, and there is no reason to uses `Array.Count` instead. The implementation of a member function is part of the VES infrastructure system, as shown in Listing A-29.

Listing A-29. Excerpt of the System.Array Implementation from Array.Length

```
public extern int Length {
    [Pure]
    [MethodImpl(MethodImplOptions.InternalCall)]
    get;
}
```

- Helper methods (member functions)
  - We can see with the implementation model of the BCL (core set) and the .NET FCL (complete set) that it is not a clever idea to put all implementation logic in a few methods (member functions). Organize units of implementation logic within the same code file or not, but in the same assembly and module file is preferable.

- If necessary, this uses unmanaged code and native code, as shown in Listing A-30.

Listing A-30. Excerpt of System.String Implementation from the CompareOrdinalIgnoreCaseHelper() unsafe Method

```
private unsafe static int CompareOrdinalIgnoreCaseHelper(String strA,
String strB)  {
            int length = Math.Min(strA.Length, strB.Length);

            fixed (char* ap = &strA.m_firstChar) fixed (char* bp =
            &strB.m_firstChar)
            {
                char* a = ap;
                char* b = bp;

                while (length != 0)
                {
                    int charA = *a;
                    int charB = *b;

                    Contract.Assert((charA | charB) <= 0x7F, "strings have
                    to be ASCII");

                    // uppercase both chars - notice that we need just one
                    compare per char
                    if ((uint)(charA - 'a') <= (uint)('z' - 'a'))
                    charA -= 0x20;
                    if ((uint)(charB - 'a') <= (uint)('z' - 'a'))
                    charB -= 0x20;

                    // Return the (case-insensitive) difference between them.
                    if (charA != charB)
                        return charA - charB;

                    // Next char
                    a++; b++;
                    length--;
                }
```

```
            return strA.Length - strB.Length;
        }
    }
```

- Native libraries
  - Remember that trustable libraries like Standard C/C++ Library, Microsoft CRT, and Microsoft UCRT are part (directly or indirectly) of any professional and advanced software such as Microsoft Windows and execution environments and their components.
  - This means that adaptable implementations of the functionalities will be present in most execution environments, directly or not, as shown in Listing A-31.

Listing A-31. Excerpt of System.String from wstrcpy()

```
internal static unsafe void wstrcpy(char *dmem, char *smem, int charCount)
{
Buffer.Memcpy((byte*)dmem, (byte*)smem, charCount * 2); // 2 used
everywhere instead of sizeof(char)
}
```

VES and Comparison operations

The equality (==) and inequality (!=) operators are used for identifying whether the instances are equal or not equal. Internally, these operators are using the implementation of the `[AnyType]->Equals()` method based on each managed type. Within the `System.Object` reference type, there is a default implementation of `System.Object.Equals()`. On the signature of the method, this default implementation declares a parameter of `System.Object`. Therefore, it is possible to inform as an argument value an instance of a reference type or an instance of a value type. Every `ref` class or value class derives directly or indirectly from `System.Object` and implicitly inherits the default implementation of `System.Object.Equals()`. When we inform as an argument value for `System.Object.Equals()` an instance of a reference type, the comparison is performed by reference. That is, the `System.Object.Equals()` method checks whether both

references are pointing to the same instance, and for this, `System.Object.Equals()` uses the `System.Object.ReferenceEquals()` static method. When we inform as an argument value an instance of a value type, it is common to think that the boxing operation is performed first because the type of parameter of `System.Object.Equals()` is the reference type `System.Object` and the base type of the informed argument value is the `System.ValueType`. Just like `System.Object`, the type `System.ValueType` is a reference type; see the declaration in Listing A-32.

Listing A-32. Declaration of System.ValueType reference type

```
/* C++/CLI (Common Language Infrastructure) projection. */

[SerializableAttribute]
[ComVisibleAttribute(true)]
public ref class ValueType abstract

/* C# programming language. */

[SerializableAttribute]
[ComVisibleAttribute(true)]
public abstract class ValueType
```

Being a reference type, the managed type automatically inherits the default implementation of the `System.Object.Equals()` method that performs the comparison by reference and not by value. However, for guaranteed performance and to keep the implementation logic dealing with values, `System.ValueType` overrides the inherited `System.Object.Equals()`. In this specialized implementation, the method applies the comparison by value, and by doing it that way, every managed type that inherits `System.ValueType` uses this specialized implementation.

To understand this, remember that to check whether two instances of reference types are equals, the VES should check whether they are in the same family of types. For example, if we are comparing two instances of a custom reference type called `Person` that directly derives from `System.Object` and it uses a specialized (overridden) implementation for the `Person.Equals()` method, the execution system checks the runtime type of the instances before choosing to call `Person.Equals()` or `System.Object.Equals()`. Value types are compared by value and family of types too. That is, when we have an instance of `System.Byte` and an instance of `System.Int32`, both with the same values, they are still different because the types are different. In these excerpts of code in C++/CLI and C# programming language, the instances of the value types

`System.Int32` an System.Byte are compared using the equality (==) operator. Both, the C# programming language and the C++/CLI do not replace the call to the equality (==) operator with a call to the implementation of the base type `System.ValueType.Equals()`. Instead, the compilers emits a sequence of CIL for specific operators. The difference in the results is that the compiler for the C# programming language, emits a sequence of CIL with the `ceq` (compare equal), and the compiler for the C++/CLI emits a sequence of CIL with the `bne.un.s`. Both CIL operators do the comparison as expected when working with value types, the difference is that the `bne.un.s` is an optimized instruction, that consumes less memory and less space in the metadata encoding, just 1-byte in each one, and can work with int8 (short-parameter form) and int32 (long-parameter form). To better understand these aspects, see Listing A-33.

Listing A-33. Explanation About Behavior of the VES for Equality and Inequality Operations

```
/* C# programming language. */

Int32 valueOne = 10;
Byte valueTwo = 10;

Boolean checkToEquals = ( valueOne == valueTwo );

/* CIL - Common Intermediate Language. */

  .locals init ([0] int32 valueOne,
           [1] uint8 valueTwo,
           [2] bool checkToEquals )

  IL_0001:  ldc.i4.s   10
  IL_0003:  stloc.0
  IL_0004:  ldc.i4.s   10
  IL_0006:  stloc.1
  IL_0007:  ldloc.0
  IL_0008:  ldloc.1
  IL_0009:  ceq
  IL_000b:  stloc.2
```

```
/* C++/CLI (Common Language Infrastructure) projection. */

Int32 valueOne{ 10 };
Byte valueTwo{ 10 };

Boolean checkToEquals{ ( valueOne == valueTwo ) };

/* CIL - Common Intermediate Language. */

.locals (   [26] uint8 valueTwo,
            [27] int32 valueOne,   [50] bool checkToEquals)

IL_0036:  ldc.i4.s   10
IL_0038:  stloc.s    valueOne
IL_003a:  ldc.i4.s   10
IL_003c:  stloc.s    valueTwo
IL_003e:  ldloc.s    valueOne
IL_0040:  ldloc.s    valueTwo
IL_0042:  bne.un.s   IL_0047
IL_0044:  ldc.i4.1
IL_0045:  br.s       IL_0048
IL_0047:  ldc.i4.0
IL_0048:  stloc.s    checkToEquals
```

For both examples, the result is true because the VES is comparing the values and not the family of type of the instances. Now, what if we opt for a direct call to the `System.ValueType.Equals()` static implementation? First, there is no static implementation for the `Equals()` method. We should remember that all value types directly or indirectly derive from the `System.ValueType` abstract reference type. After learning about the `System.ValueType.Equals()` method, we understand the following:

- It is an instance method.
- It is an inherited instance method of `System.Object`.
- It is overridden by the `System.ValueType abstract` reference type.

Listing A-34 shows the declaration.

Listing A-34. Declaration of the System.ValueType.Equals Instance Method

```
/* C++/CLI (Common Language Infrastructure) projection syntax. */

public:
virtual bool Equals(
    Object^ obj
) override

/* C# programming language syntax. */

public override bool Equals(
    Object obj
)
```

As shown in Listing A-35, when the compiler looks for a sequence like ones in the following examples, it verifies that it is necessary to support some syntactical sugar to avoid syntactical and semantic errors. In fact, to support this expression by calling an instance method, without an explicit instance, the generated intermediate code will fill the gaps. Both lines with box keywords will put each of the instances of value types encapsulated within instances of the `System.Object` reference type. So, we have two instances of reference types on the top of evaluation stack, and when the intermediate instruction call is performed, we have a comparison based on the family of types and not by value (bit-by-bit).

Listing A-35. CIL for the Call of the System.ValueType.Equals() Instance Method

```
/* C# programming language sample code. */

Int32 valueOne = 10;
Byte valueTwo = 10;

Boolean checkToEquals = ValueType.Equals( valueOne,  valueTwo );

/* CIL - Common Intermediate Language. */

  .locals init ([0] int32 valueOne,
           [1] uint8 valueTwo,
           [2] bool checkToEquals )
```

```
  IL_0001:  ldc.i4.s   10
  IL_0003:  stloc.0
  IL_0004:  ldc.i4.s   10
  IL_0006:  stloc.1
  IL_0007:  ldloc.0
  IL_0008:  box        [mscorlib]System.Int32
  IL_000d:  ldloc.1
  IL_000e:  box        [mscorlib]System.Byte
  IL_0013:  call       bool [mscorlib]System.Object.Equals(object, object)
  IL_0018:  stloc.2
/* C++/CLI (Common Language Infrastructure). */

Int32 valueOne{ 10 };
Byte valueTwo{ 10 };

Boolean checkToEquals{ ValueType.Equals( valueOne, valueTwo ) };

/* CIL - Common Intermediate Language. */

  .locals (uint8 valueTwo,
           int32 valueOne,
           bool checkToEquals)
  IL_0000:  ldc.i4.s   10
  IL_0002:  stloc.1
  IL_0003:  ldc.i4.s   10
  IL_0005:  stloc.0
  IL_0006:  ldloc.1
  IL_0007:  box        [mscorlib]System.Int32
  IL_000c:  ldloc.0
  IL_000d:  box        [mscorlib]System.Byte
  IL_0012:  call       bool [mscorlib]System.Object.Equals(object, object)
  IL_0017:  stloc.2
```

With these examples using `System.ValueType.Equals()`, the result is false. With values types, we should be aware of boxing/unboxing. Boxing/unboxing is a scenario of last resort to deal with the question of polymorphism, which is one of the

principles of object-oriented programming. However, in this example, every value type inherits directly from the `System.ValueType` reference type, which overrides the inherited implementation of `System.Object.Equals()`. So, every value type that inherits that specialized implementation of `System.ValueType.Equals()` considers as equals the instances that are of the same managed type and that represent the same value (sequence of bits). For a value type, all field members are also value types; the comparison is made byte by byte (internally using the `memcmp` function from Microsoft UCRT/Microsoft CRT/Standard C/C++ Library). If any field member in a value type instance is a reference type, the Reflection technology is used to inspect the value types and reference types and do the comparisons for the values using the most derived `Equals()` method implementation of each managed type. With our custom managed types derived from `System.ValueType`, we should override the `System.ValueType.Equals()` method and create a specialized implementation whose signature has the parameter of our custom type. The value types on the BCL (core set) and the .NET FCL (complete set) are implemented in that way. See the declaration of `System.Int32`, which is a value type in Listing A-36.

Listing A-36. Declaration of System.Int32 Value Type

```
/* C++/CLI (Common Language Infrastructure) projection. */

[SerializableAttribute]
[ComVisibleAttribute(true)]
public value class Int32: IComparable, IFormattable, IConvertible,
IComparable<Int32>, IEquatable<Int32>

/* C# programming language. */

[SerializableAttribute]
[ComVisibleAttribute(true)]
public struct Int32: IComparable, IFormattable, IConvertible,
IComparable<Int32>, IEquatable<Int32>

/* C++/CLI (Common Language Infrastructure) projection. */

public:
      virtual Boolean Equals( Object^ obj ) override // overrides
      System.ValueType.Equals().
```

```
      virtual Boolean Equals( Int32 obj ) sealed // implicit implementation
      of System.IEquatable<Int32>.Equals().

/* C# programming language. */
public override Boolean Equals( Object obj ) // overrides
System.ValueType.Equals().
public Boolean Equals( Int32 obj ) // implicit implementation of
System.IEquatable<Int32>.Equals().
```

Using the Equality and Inequality Operators vs. Explicit Calls to an Equals() Method

The next three examples help us to better understand the use of equality and inequality operators and the use of the `Equals()` method.

The first example uses the equality (`==`) operator, the second example explicitly calls for an implementation `Equals()` method, and the third example does a casting for determining which implementation of the `Equals()` method should be called. The source code is simple; it's only 30 lines of source code. It is recommended to use a console type application because it is easy to read the CIL generated by the compiler. It is interesting to remove all references for assemblies `System.*` and others that Microsoft Visual C# and Microsoft Visual C++ templates automatically include and keep only `mscorlib` and `System` references.

In this first example, the instances are compared byte by byte and not by the implementation of the `System.Int32.Equals()` method. If we uncomment the line that uses `Int32.Equals()`, the most derived implementation is used. In the case of the implementation of the `System.Int32` value type, it is the `System.IEquatable<Int32>.Equals( Int32 )` method implementation. When we must guarantee that a specific implementation of a method should be used, it is recommended to use casting, which explicitly indicates this.

Listing A-37. Full Code in C++/CLI Projection and C#

```
/* C++/CLI (Common Language Infrastructure) projection. */

Int32 value1{ 12 };
Int32 value2{ 12 };
Boolean result;

/*result = ( value1 == value2 );
result = value1.Equals( value2 );
result = ( ( ( IEquatable<Int32>^ ) value1 )->Equals( value2 ) ); */

Console::ReadLine();

DateTime finish;
DateTime start{ DateTime::Now };

for( UInt32 index{}; ( index != 10000000 ); index++ ) result = ( value1 ==
value2 );
/* for( UInt32 index{}; ( index != 10000000 ); index++ ) result = value1.
Equals( value2 );
for( UInt32 index{}; ( index != 10000000 ); index++ ) result = ( ( (
IEquatable<Int32>^ ) value1 )->Equals( value2 ) ); */

finish = DateTime::Now;

Console::ReadLine();

StringBuilder^ buffer{ gcnew StringBuilder };
buffer->AppendFormat( "Start-> Minutes: {0} Seconds: {1} Milliseconds:
{2}\n", start.Minute.ToString(), start.Second.ToString(), start.
Millisecond.ToString() );
buffer->AppendFormat( "Finish-> Minutes: {0} Seconds: {1} Milliseconds:
{2}\n", finish.Minute.ToString(), finish.Second.ToString(), finish.
Millisecond.ToString() );

Console::WriteLine( buffer->ToString() );

buffer->Clear();
buffer = {};
```

```
/* C# programming language. */
Int32 value1 = 12;
Int32 value2 = 12;
Boolean result;

/* result = ( value1 == value2 );
result = value1.Equals( value2 );
result = ( ( ( IEquatable<Int32> ) value1 ).Equals( value2 ) ); */

Console.ReadLine();

DateTime finish;
DateTime start = DateTime.Now;

for (UInt32 index = 0; ( index != 10000000 ); index++) result = ( value1 ==
value2 );
/* for (UInt32 index = 0; ( index != 10000000 ); index++) result = value1.
Equals(value2);
for (UInt32 index = 0; ( index != 10000000 ); index++) result = ( ( (
IEquatable<Int32> ) value1 ).Equals( value2 ) ); */

finish = DateTime.Now;

Console.ReadLine();

StringBuilder buffer = new StringBuilder();
buffer.AppendFormat("Start-> Minutes: {0} Seconds: {1} Milliseconds:
{2}\n", start.Minute.ToString(), start.Second.ToString(), start.
Millisecond.ToString());
buffer.AppendFormat("Finish-> Minutes: {0} Seconds: {1} Milliseconds:
{2}\n", finish.Minute.ToString(), finish.Second.ToString(), finish.
Millisecond.ToString());

Console.WriteLine( buffer.ToString() );

buffer.Clear();
buffer = null;
```

But the sequence of CIL is not the same for the three examples.

With this first example, we are using the equality (==) operator. **`bne.un.s`** is used to transfer the execution to the local with intermediate instructions that determine that the zero value must be loaded and assigned as a result. The zero values denote the value false in an instance of value type `System.Boolean`, and any value different from zero denotes the value true. The instructions **`ldc.i4.0`** and **`ldc.i4.1`** stand for "load numeric constant." In this case, the instances of `System.Int32` are stack allocated, one instance for the value 0 and the other instance for the value 1. The intermediate instruction **`br.s`** means "unconditional branch." That is, unconditionally transfer to the indicated local. These are optimized instructions for bit pattern comparisons, and they do not generate calls for the `System.ValueType.Equals()` (`System.Object.Equals()` override), `System.Int32.Equals()` (`System.IEquatable<T>` explicit implementation), and `System.Object.Equals()` methods. An example of this is shown in Listing A-38.

Listing A-38. Call to the Equality Operator

```
Boolean result{ ( value1 == value2 ) };

  IL_0023:  ldc.i4.s   12
  IL_0025:  stloc.s    value1
  IL_0027:  ldc.i4.s   12
  IL_0029:  stloc.s    value2
  IL_002b:  ldloc.s    value1
  IL_002d:  ldloc.s    value2
  IL_002f:  bne.un.s   IL_0034
  IL_0031:  ldc.i4.1
  IL_0032:  br.s       IL_0035
  IL_0034:  ldc.i4.0
  IL_0035:  stloc.s    result
```

The second example is the result of an explicit call for the implementation of the `System.Int32.Equals()` method. The highlighted line shows the call intermediate instruction, and the instance keyword indicates the most derived implementation of the `System.Int32.Equals()` method. The intermediate call designates that a call for a method must be performed, but the instance keyword does not belong to the call intermediate instruction; it is contextual information that is used by the VES. In this case, it is part of the metadata about the method, and that is sufficient for the execution engine

on the platform to understand how this method should be called. In this example, this information tells the execution engine that this method is an implementation that belongs to type `System.Int32` and not to some ancestor, such as `System.ValueType`. Just as the use of the inequality operator is more efficient than this explicit call to the `System.Int32.Equals()` method presented in this second example, we will see in the sequence with the third example that the casting to interface `System.IEquatable<Int32>` is less efficient than these first two examples. An example of this is shown in Listing A-39.

Listing A-39. Call to the System.Int32.Equals() method

```
Boolean result{ ( value1.Equals( value2 ) ) }; // Implicit implementation
of System.IEquatable<Int32>.Equals().

  .locals ([36] int32 V_36 )
  IL_0023:  ldc.i4.s   12
  IL_0025:  stloc.s    value1
  IL_0027:  ldc.i4.s   12
  IL_0029:  stloc.s    value2
  IL_002b:  ldloc.s    value1
  IL_002d:  stloc.s    V_36
  IL_002f:  ldloca.s   V_36
  IL_0031:  ldloc.s    value2
  IL_0033:  call       instance bool [mscorlib]System.Int32.Equals(int32)
  IL_0038:  stloc.s    result
```

The third example introduces the use of casting to `System.IEquatable <Int32>` and the use of the `Equals` method. The statement `ldloc.s value1` loads the instance of `System.Int32` in the stack. Next is the statement box `[mscorlib] System.Int32`, which performs boxing with this instance of `System.Int32`. Note that casting is applied on the instance indicated by `value1`, so boxing is performed from the instance that `value1` represents in the code. The `callvirt` intermediate instruction performs the call to a virtual or nonvirtual method. In this example, it is a virtual method, in other words, an implementation of the `Equals` method. It is important to remember that there are four subtypes for a reference type, and the interface type is one of these four subtypes. When the `callvirt` intermediate instruction is used, this indicates that a search in the type hierarchy is performed to properly identify which implementation the code refers to. When a type is declared, the list of interfaces that this method implements is part of the

ancestor group, together with a base class, implicitly `System.Object` or some other class included in the declaration.

So, the list of ancestors includes a base class and N interfaces. Because classes and interfaces are reference types, they inherits only from reference types and not from some value type. Because the base class for all and any managed type is the reference type `System.Object`, when boxing is performed, a copy of the value type is assigned to an instance of `System.Object`. When the `callvirt` intermediate instruction is used, a search for the nearest ancestor that provides an implementation for that method is performed. In addition to using the metadata, the location of the method implementation is performed at runtime and is not determined at compile time. In the `System.Int32` hierarchy, the nearest ancestor, by definition, is the reference type `System.ValueType`, but the `System.ValueType` does not have an implementation for `System.IEquatable<T>`. In the sequence, depending on the casting, the nearest ancestor is `System.IEquatable<T>`, but being an interface type, the type does not have implementation but determines that the type that declared the interface type is an ancestor and has an implementation for its members. The value type `System.Int32` is checked, and an implementation for `System.IEquatable<T>.Equals()` is found, being in accordance with the other rules for the call to be performed.

In this example, a less efficient path was used to perform a comparison recognized and optimized directly in the CLR platform implementation. If you carefully observe the CIL instructions in the remainder of the program, you do not find a unbox intermediate instruction. The instance of `System.Object`, which was created automatically to temporarily store the copy of the values of the instance of value type `System.Int32`, is persisted until the next pass of the GC. Remember that components such as the GC have distinct implementations for Microsoft Windows variations, such as Microsoft Windows 10 and Microsoft Windows Server 2016. An example of this is shown in Listing A-40.

Listing A-40. Explicit Call of the System.IEquatable<Int32>.Equals() Method Implementation

```
Boolean result{ ( ( ( IEquatable<Int32>^ ) value1 )->Equals( value2 ) ) };

  IL_001d:  ldc.i4.s   12
  IL_001f:  stloc.s    value1
  IL_0021:  ldc.i4.s   12
  IL_0023:  stloc.s    value2
```

```
IL_0025:  ldloc.s    value1
IL_0027:  box        [mscorlib]System.Int32
IL_002c:  ldloc.s    value2
IL_002e:  callvirt   instance bool class [mscorlib]System.
IEquatable`1<int32>.Equals(!0)
IL_0033:  stloc.s    result
```

Index

Symbols

A

R. Villela, *Pro .NET Framework with the Base Class Library*, https://doi.org/10.1007/978-1-4842-4191-2

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